CECOM

CENTER FOR SOFTWARE ENGINEERING

ADVANCED SOFTWARE TECHNOLOGY

Subject: - **IMPACT OF DOMAIN ANALYSIS ON REUSE METHODS**

Final Report

CIN: **C04-087LD-0001-00**

6 NOV 1989

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	6 NOV 89	Final Report

4. TITLE AND SUBTITLE

Impact of Domain Analysis on Reuse Methods

5. FUNDING NUMBERS

C- DAAL03-86-D-0001

6. AUTHOR(S)

Kathleen A. Gilroy, Edward R. Comer, J. Kaye Grau, Patrick J. Merlet

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Software Productivity Solutions, Inc.
P.O. Box 361697
Melbourne, FL 32936

8. PERFORMING ORGANIZATION REPORT NUMBER

1138

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

88350

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The purpose of this effort is to analyze the domain process and its relationship to the development and use of reusable components. It includes an identification of the critical issues and risks involved throughout the process. Capabilities of automated tools which could be used to perform various aspects of a domain analysis are also investigated. Finally, it provides a set of guidelines for conducting a domain analysis for embedded systems.

14. SUBJECT TERMS

Domain Analysis, Reuse Methods, Software Components

15. NUMBER OF PAGES

223

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to **stay within the lines to meet optical scanning requirements.**

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD	- See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	- See authorities.
NASA	- See Handbook NHB 2200.2.
NTIS	- Leave blank.

Block 12b. Distribution Code.

DOD	- DOD - Leave blank.
DOE	- DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA	- NASA - Leave blank.
NTIS	- NTIS - Leave blank.

Block 13. Abstract. Include a brief (Maximum 200 words) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (NTIS only).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

IMPACT OF DOMAIN ANALYSIS ON REUSE METHODS:

Final Report

PREPARED FOR: U S ARMY CECOM
CENTER FOR SOFTWARE ENGINEERING
AMSEL-RD-SE-AST
FORT MONMOUTH, NJ 07703

PREPARED BY: Software Productivity Solutions, Inc.
122 North 4th Avenue
Indialantic, FL 32903

6 NOV 1989

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

List of Figures

List of Figures (continued)

List of Tables

EXECUTIVE SUMMARY

The Government is faced with an urgent need to increase productivity in the development and maintenance of mission critical computer systems. Reusing software across multiple application systems is one way to accomplish this goal. Recent research in the area of reusable software indicates that *domain analysis* is the first activity which should be performed during the development of reusable software. A domain analysis identifies commonalities between systems within a given problem domain. These commonalities (typically represented as objects, operations and relationships which characterize the domain -- a domain model) are then implemented as software components which can be reused by new systems within that domain.

Few domain analyses have been done to date, largely due to the complexity of the problem and the expense of the process. Well-defined methods for performing domain analyses and for measuring their "goodness" do not yet exist. The immaturity of formal methods is the probable cause for the almost total lack of domain analysis tools.

The primary objective of this effort is to develop an approach to make domain analysis practical and effective for the development of reusable software and the reuse of that software in new application systems. The main focus is domain analysis within the context of DoD software development; more specifically, the interest is in embedded Ada software for Army applications.

The initial work performed under this effort is a survey of existing and emerging methods and tools for performing a domain analysis and applying its results. Based on the benefits and shortcomings of existing approaches, alternative approaches to domain analysis for Army Ada applications are analyzed, and the most promising selected for further development. A consistent, cohesive, and complete methodology for domain analysis which addresses the major issues is presented.

The postulated methodology for domain analysis is based on an object-oriented paradigm. A three phased approach is recommended for domain analysis: (1) Model the domain, (2) architect the domain, and (3) develop software component assets. A new concept introduced is adaptation analysis, i.e., the identification of differences among application systems in the domain.

Automated capabilities which support the domain analysis process are proposed. These capabilities address the application of existing tools to domain analysis, as well as future tool developments. This effort identifies and addresses the key technical areas which affect the automation of domain analysis. These areas include knowledge acquisition and knowledge-based guidance, domain languages and language-based processing, information models and data storage and retrieval, and tool and environment integration.

The primary conclusion of this research is that domain analysis, when done right, is a significant undertaking yet produces a significant benefit.

The results of the method, tool and issue analyses are summarized and organized into guidelines for domain analysis. As one of the first significant and comprehensive research efforts in domain analysis, the report identifies where tool prototyping, methods experimentation, and additional technical research are needed.

This work was performed for the U.S. Army Communications Electronics Command, (CECOM), Center for Software Engineering, under subcontract to Battelle, Scientific Services Program, contract number DAALO3-86-D-0001, Delivery Order 1138.

1.0 INTRODUCTION

1.1 Problem Statement

The Government is faced with an urgent need to increase productivity in the development and maintenance of mission critical computer systems. Reusing software across multiple application systems is one way to accomplish this goal.

Recent research in the area of reusable software indicates that *domain analysis* is the first activity which should be performed during the development of reusable software. A domain analysis identifies commonalities between systems within a given problem domain. These commonalities (typically represented as objects, operations and relationships which characterize the domain -- a domain model) are then implemented as software components which can be reused by new systems within that domain. According to the developers of the Common Ada Missile Packages (CAMP) software, to "attempt to start a software reusability program without adequately performing this [domain] analysis is as foolish as attempting to design a software system without performing an analysis of the software requirements." [MCN86a]

Few domain analyses have been done to date, largely due to the complexity of the problem and the expense of the process. The earliest attempts have been in mature, stable application areas like business systems (although they did not call the process domain analysis) [LAN79]. Belady bemoans the "difficulty of finding the hidden commonality of functions across many applications. The only way to do this is by tediously studying complex and often structurally obsolete software--not considered a pleasant or even respectable activity today." [BEL79] Not all researchers agree that existing systems comprise the only, even the primary, input to the construction of a domain model, but deriving such a model does require a more extensive and difficult analysis than is performed on most system developments.

Given the increased cost associated with performing a domain analysis, it is imperative that there is a return on the investment. The potential negative effects a "bad" domain analysis has on developing systems in the domain also increases the risks of a reuse effort. A bad domain analysis could result in a range of problems, from not being able to find a required component, to increasing the effort to reuse a component, to ending up with a set of components that can't be reused at all. According to the developers of Draco, a domain-oriented program generation system, it is "easy to construct a bad domain and very hard to construct a good one." [NEI83]

Well-defined methods for performing domain analyses and for measuring their "goodness" do not yet exist. Recent reuse efforts have incorporated ad hoc approaches to domain analysis. Little information on the advantages and drawbacks of various approaches is available. In his survey of domain analysis, Prieto-Diaz found that "all reported experiences concentrate on the outcome, not on the process." [PRI88] While some researchers have attempted to formalize the process, there are many conflicts concerning what a "domain" is, what a "domain analysis" entails, and what kinds of "reusable components" a domain analysis supports.

The immaturity of formal methods is probable cause for the almost total lack of domain analysis tools. Prieto-Diaz notes that it "is difficult to specify tools for a process we do not understand fully." [PRI88] Although there are some tools which support portions of the analysis process, no tools exist which automate the entire process. Although unique tools are beginning to be identified, current emphasis seems to be toward applying existing analysis tools to the domain analysis process.

Domain analysis is only one part of the process of developing and reusing software. The domain analysis process must be integrated with the other activities performed as part of systems and software development. McCain warns that if "the software industry cannot adequately establish software development approaches that emphasize the construction of reusable software components, then attempts to reuse software from a component library will, of course, prove to be futile." [MCC85]

Evidence indicates that 15 to 50 percent of new embedded applications could be comprised of reusable software [SPS88]. Domain analysis is expected to play a critical role in the Army's ability to meet this potential, but significant research and development will be required to produce effective methods and tools.

1.2 Objectives and Approach

The primary objective of this effort is to develop an approach to make domain analysis practical and effective for the development of reusable software and the reuse of that software in new application systems. The main focus is domain analysis within the context of DoD software development; more specifically, the interest is in embedded Ada software for Army applications. This objective is addressed through the development of a preliminary, but comprehensive, set of guidelines for performing state-of-the-art domain analyses. The guidelines assist software developers and acquisition personnel in applying domain analysis to achieve reuse goals. The guidelines also suggest areas where future automation efforts might be directed.

Another objective of this effort is to reinforce the importance of domain analysis as part of a software reuse program. A particular goal is to assist in avoiding the potential negative effects of ad hoc or otherwise "bad" domain analyses. However, rather than discouraging attempts to use domain analysis, the goal is to encourage the experimentation and feedback necessary to advance in this new but critical technology area.

Another objective of this effort is to summarize and coalesce the varied and dispersed information on domain analysis into a single source of reference. Existing and emerging methods, languages, tools and techniques that have been or could be applied to the domain analysis process are examined, and a snapshot of research results is provided.

As with all emerging technology, any attempt to apply domain analysis presents many issues and risks. Although most of them are technical there are sociological and economic problems to be dealt with as well. This report begins to identify the critical issues and risks associated with the domain analysis process and postulate alternatives for resolving those issues and dealing with those risks. Cost/benefit tradeoffs are analyzed and evaluated, and specific recommendations made where possible. Initial issue areas are the impacts of methods, tools, languages, personnel, organizational structure, policy, and application domain on the domain analysis process. Other important issues which should be addressed are evaluation and validation of domain specifications and maintenance of those specifications.

The initial work performed under this effort is a survey of existing and emerging methods and tools for performing a domain analysis and applying its results. This survey addressed the following issues:

- o Strategies and paradigms for domain analysis
- o Process models for domain analysis (including relationships to the overall process of software reuse)

o Inputs to domain analysis activities

o Methods for performing domain analysis activities

o Products of domain analysis activities

o Personnel supporting domain analysis activities

o Tools supporting domain analysis activities

Based on the benefits and shortcomings of existing approaches, alternative approaches to domain analysis for Army Ada applications are analyzed, and the most promising selected for further development. A consistent, cohesive, and complete methodology for domain analysis which addresses the major issues is presented.

Automated capabilities which support the domain analysis process are proposed. These capabilities address the application of existing tools to domain analysis, as well as future tool developments. This effort identifies and addresses the key technical areas which affect the automation of domain analysis. These areas include knowledge acquisition and knowledge-based guidance, domain languages and language-based processing, information models and data storage and retrieval, and tool and environment integration. Ideally, automation mechanizes methodology (as opposed to defining methods based on available tools). As such, the definition of domain analysis methods is completed before proposing tools to support the method.

The results of the method, tool and issue analyses are summarized and organized into guidelines for domain analysis. As one of the first significant and comprehensive research efforts in domain analysis, the report identifies where tool prototyping, methods experimentation, and additional technical research are needed.

1.3 Overview of Report

This report is organized into four sections and three appendices. Section 1, Introduction, describes the problem addressed by this effort, the specific objectives and approach taken to solve the problem, and provides this overview of the document organization and contents.

Section 2, Survey of Domain Analysis Approaches, summarizes the results of a survey of domain analysis research and case histories. Criteria which were evaluated include: strategies and paradigms for domain analysis, domain analysis process models, inputs to the domain analysis process, methods for performing domain analysis activities, products of domain analysis activities, personnel for performing domain analysis activities, and tools supporting domain analysis.

Section 3, Guidelines for Conducting a Domain Analysis, describes a proposed methodology for domain analysis, identifies supporting tools, and provides guidance in dealing with issues and risks of domain analysis.

Section 4, Conclusions and Recommendations, summarizes the results of this effort, identifies areas requiring further research and development, and makes specific recommendations for future domain analysis efforts.

Appendix A provides a glossary of acronyms. Appendix B provides a list of references. Appendix C provides synopses of recent research efforts into domain analysis which were used in developing the main document.

2.0 SURVEY OF EXISTING AND EMERGING APPROACHES TO DOMAIN ANALYSIS

This section summarizes existing and emerging approaches to domain analysis as represented by written documentation of efforts in the area. Detailed synopses of this documentation are provided in Appendix C.

The approaches are assessed and compared according to the following categories of evaluation topics:

- o Strategies and paradigms for domain analysis
- o Process models for domain analysis
- o Inputs to domain analysis activities
- o Methods for performing domain analysis activities
- o Products of domain analysis activities
- o Personnel supporting domain analysis
- o Tools supporting domain analysis

The documents were analyzed using sets of questions related to each evaluation category. Lessons learned, issues and risks, and problems encountered in each of the areas were also identified. The results were coalesced into summaries and comparisons of domain analysis approaches with respect to each area, and are presented in the following sections.

2.1 Strategies and Paradigms for Domain Analysis

General strategies and paradigms that have been used or are proposed for perfonning domain analyses are addressed by the following questions:

- o What rationale is offered for performing domain analysis? How does the approach assist in meeting reuse goals and objectives?
- o How does domain analysis relate to the overall approach for software reuse? To the overall approach for systems and software development?
- o What is the overall approach for domain analysis? What factors influenced the decision to adopt this particular approach?

The primary thrust of any reuse effort is to increase the productivity of application developments. The purpose of most current domain analysis efforts is to further increase the productivity potential provided by reuse. Specific goals and benefits of domain analysis are discussed in section 2.1.1.

Section 2.1.2 defines an overall approach to software reuse and describes how domain analysis has or can fit into the reuse process. This section includes a discussion of the relationship of reuse and associated domain analysis activities to phased system developments.

Different paradigms for software reuse impose constraints on the approach that must be taken for performing a domain analysis. Section 2.1.3 defines three reuse paradigms and how these paradigms have or can influence the approach to domain analysis.

Section 2.1.4 identifies organizational strategies for developing reusable software and how these might affect a domain analysis effort.

Different domain analysis approaches placed different emphasis on the use of existing systems as an input to the domain analysis process. Section 2.1.5 identifies three overall strategies for domain analysis based on this factor.

2.1.1 Goals and Benefits of Domain Analysis

Improved productivity in reusing software for application development is the reason most generally cited for applying domain analysis. Improved productivity can result in several ways.

A domain analysis can improve the reusability of new components which are developed. Components derived from a domain analysis are more likely to be usable in building new application systems in that domain. Most of the surveyed research focussed on this benefit of domain analysis.

A domain analysis usually results in the *explicit* specification of some form of domain information, and information which is explicit can be reused. "The reuse of analysis information... is the most powerful sort of reuse." [PRI88] The possible benefits of this reuse will depend on the amount and type of information which is recorded during the domain analysis.

A domain analysis can result in a *reusable specification* of application systems in that domain. Application-specific versions of the domain specification can subsequently be generated for less effort. At an extreme, program generation technology could be used to automatically derive new system instances directly from the domain-oriented specification.

Domain analysis information can be used to *verify* independently developed requirements and specifications for new application systems in the domain. "Since the cost of correcting an error grows very rapidly as a system's life cycle progresses, the ability to detect specification problems early may be a primary benefit of domain analysis." [NEI88] Iscoe also promotes the reuse of application domain knowledge to avoid specification errors. [ISC88]

One working group recommended that domain models be used to *educate* new personnel about the general structure and operation of systems in the domain. [NEI88] This could make new personnel more rapidly productive. Existing personnel would presumably already be more productive through the use of common domain models.

2.1.2 Relationship of Domain Analysis to Software Reuse

The reuse of software involves three distinct software development processes [SPS87]:

1. The process of developing reusable components

2. The process of reusing components

3. The process of developing application systems/software

(Note: The reader should loosely interpret the term "reusable component" as used here to be *any* reusable software entity. Specific kinds of reusable software entities are mentioned throughout this document.)

The process of developing reusable components involves *identifying* the reusable components to be developed, *developing* the components, *validating and verifying* the components, *classifying* the components, *storing* the components, and *maintaining* the components. Bailin calls these *supply side* activities [BAI88], while the Reuse Tools and Environments Working Group (RTEWG) calls these *domain development* activities [RTE88]. The result of this process is what Arango refers to as the *reuse infrastructure* [ARA87]. Proposed life cycles for reusable software are generally very similar to the familiar phased approach to systems/software development.

The process of reusing components involves: *recognizing* the potential for reuse of a component, *selecting* a component to reuse, *synthesizing* the component into the system under development, and *validating and verifying* the component in its new context. Component synthesis may involve adaptation of either the component or application system as part of integrating the component into the new system. For Bailin, these are *consumer side* activities [BAI88], the RTEWG calls these *domain integration* activities [RTE88], while Arango considers them part of *reuse mechanization* [ARA87].

For our purposes, the process of developing application systems/software includes systems level activities, conforms to the requirements of DOD-STD-2167A [DOD88], and utilizes the Ada language for implementation [ARM83]. A portion of the application software is developed using reusable components, and the remainder is developed from scratch. An important aspect of DOD-STD-2167A is its allocation of development activities to system and software development phases. Figure 2.1.2-1 shows a typical relationship between the phases of the system and software development cycles.

Ideally, all of the activities mentioned above are integrated into a comprehensive, consistent, and effective process for software reuse. Unfortunately, a methodology which fully accomplishes this objective has yet to be developed. It is our assessment that domain analysis concepts and techniques can provide much of the "glue" that is needed to accomplish the integration.

The activities and products of domain analysis are potentially applicable in varying degrees and combinations to all three processes of software reuse. Based on the survey, however, there are certain activities within each reuse process to which domain analysis has been most commonly applied or recommended.

With respect to the process of developing reusable components (reuse process 1 above), the *process* of domain analysis is most commonly applied to identifying the reusable components to be developed, and classifying the components that are developed. The domain analysis results in an overall model shared by all systems in an application domain. The pieces of this model *are or will become* the reusable components used by applications in the domain. The structure of this model and the languages used to express the model can be used in classifying the components in a domain-specific way.

With respect to the process of reusing components (reuse process 2 above), the *products* of a domain analysis are most commonly used in selecting a component to reuse in a new application system, and in adapting components during synthesis into a new system. A new application system can be thought of as an instance of the domain model. The domain classification structure and other component documentation assists the reuser in finding and discriminating among components that could be used in building the new

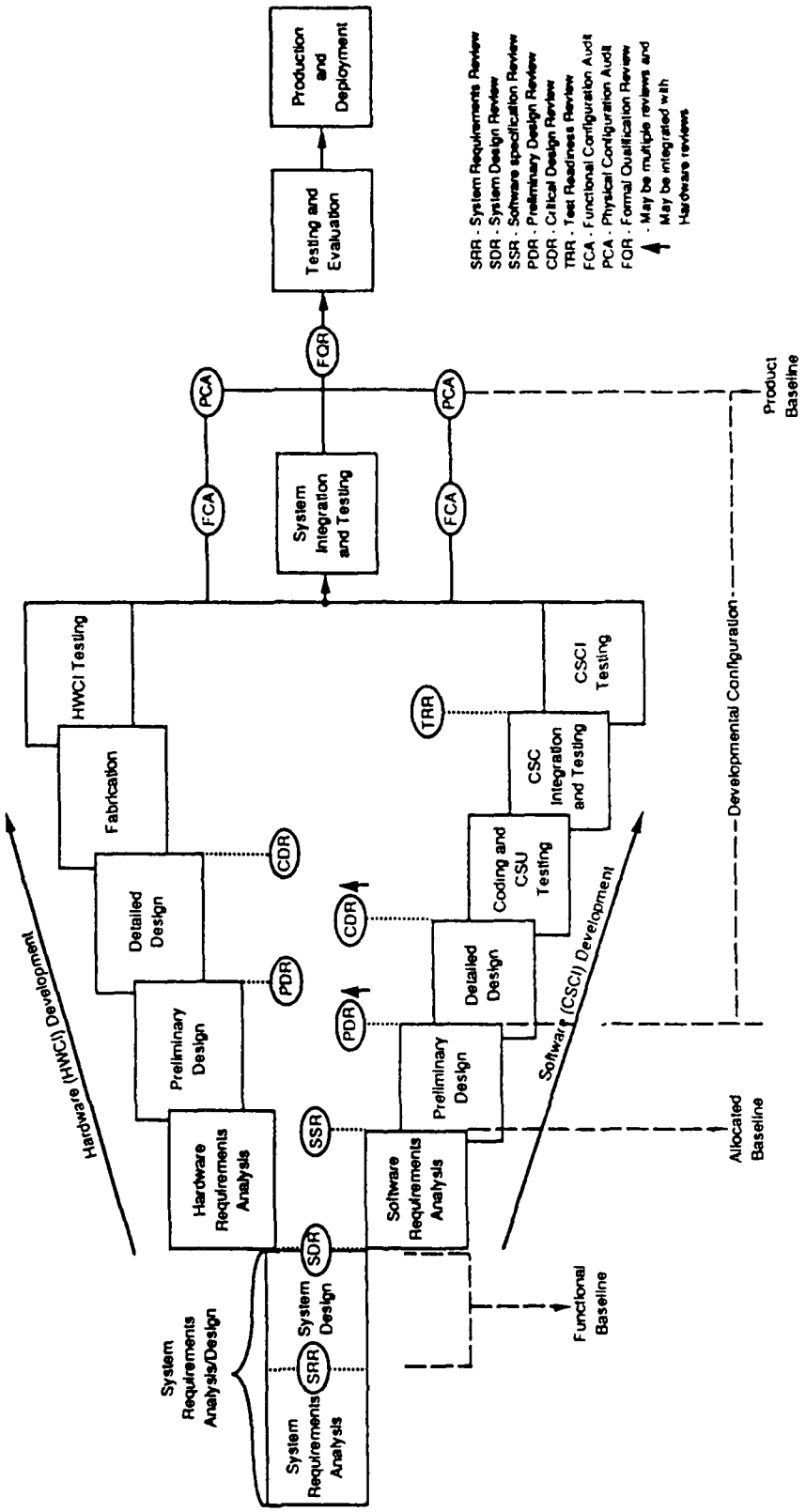


Figure 2.1.2-1. Typical System/Software Development Cycle [DOD88]

system. Domain knowledge can be used to guide the reuser in adapting a component to meet application-specific requirements.

With respect to the process of developing new application systems/software (reuse process 3 above), the *process* of domain analysis can influence the methods and tools used to build new application software, and the *products* can influence the end software itself. Compatibility of the domain analysis methodology with the application development methodology is important because domain analysis generally occurs "prior to systems analysis and [its] output (i.e., a domain model) supports systems analysis in the same way that systems analysis output (i.e., requirements analysis and specifications document) supports the systems designer's tasks." [PRI87] Depending on the level of abstraction of the domain, compatibility with methodologies used at other phases of an application's life cycle can be critical. For example, consider the domain of "data structures", a low-level application-independent domain. Specification of the requirements and design of components in this domain typically does not occur until well into the software design phase of large systems.

2.1.3 Impact of Reuse Paradigms

Reuse paradigms address what constitutes a reusable component, and how it is used to build new application systems. Choice of a particular paradigm can influence the strategy taken for domain analysis.

Arango coins the term *model of the reuse task* to refer to the reuse paradigm. He identified the following three distinct models of reuse which appear to represent the full range of existing or proposed reuse strategies [ARA87]:

- o *Construction of systems using program components*, where the reused components actually become part of the new software system. This is also known as a *parts composition* approach.

- o *Generation of systems using pattern components*, where the reused components are plans for creating programs. A specification for the new software system is transformed into a program using the plans. This process is often called *software synthesis*.

- o *Generation of systems using process components*, where the reused components are goals for creating programs. Plans for creating programs are generated based on the relationships of the new system specification to the goals. This approach is also called a *planning paradigm*.

Arango, like most of the research surveyed, chose the first approach as the basis for further developing his reuse framework and methods. The Draco paradigm is representative of the second approach. [NEI83] Huff and Lester describe a system that supports the third approach. [HUF87]

Some efforts employ a combination of constructive and generative approaches. The CAMP developers described a component construction approach overall, but also included a variation of software generation using component schematics. [MCN86a] The Synthesis Project at the Software Productivity Consortium also supports a primarily constructive approach with some automated component selection and adaptation capabilities. [PYS88]

Generation approaches support a consistent domain analysis paradigm throughout the software development process. Initially starting with a *problem domain*, the system designer deals with various *modeling domains* during the transformation and refinement

process until the final *target domain* is reached. "The concept of domain here is very useful in supplying a psychological set to the system's designer (i.e., the designer must only consider and think about the objects and operations of one domain at a time)." [NEI83] Components at any level of domain abstraction are potentially reusable. Neighbors even defines an explicit step in the domain analysis process for searching out existing objects to reuse in the definition of new objects. [NEI83]

Planning systems take the program generation approach a step further. This approach requires the formalization of the three *processes* of reuse (component development, component reuse, and application development). For example, domain analysis itself would be a domain to be analyzed and specified. The domain analysis process goals and the rules used to constrain the generation of a domain analysis process plan could be adapted as needed to accommodate the needs of specific application domains, development methods, etc. While process modeling is not a new research topic, its actual application to the automatic generation of software systems is not yet mature enough to consider as a viable approach for domain analysis.

Constructive approaches separate the definition of the domain model from the development of the reusable components. The domain analysis process and products are generally less formal than those associated with generation approaches. Further, while generation approaches require the explicit definition of transformation rules between domains, deriving and maintaining mappings between domain models and reusable components is not generally performed with constructive approaches.

For most of the constructive approaches surveyed, the reusable components are source code units. The term *reusable encapsulation* was coined by Arango [ARA87] and adopted by Prieto-Diaz [PRI87] to refer to this kind of component.

Several efforts employing constructive approaches propose the development and reuse of domain models representing the architecture of application systems. SofTech calls such models *generic architectures* [QUA88], and the Software Productivity Consortium calls them *canonical designs* [PYS88]. Architectural domain models are at a level of abstraction between *problem domain* models and *reusable encapsulations*.

2.1.4 Organizational Strategies

The Software Engineering Institute identified three organizational approaches for developing reusable software, and compared their ability to support a reuse program. [SEI88b] The following list paraphrases some of their findings in terms more directly related to domain analysis:

- *A group formed as a project spin-off effort* typically does not perform a domain analysis, but may address domain-level considerations in later repackaging of project-specific components. The results are not very likely to be usable on other projects.

- *An autonomous parts group* performs domain analyses independent of any specific project. The results are likely to support multiple projects, but may fail to support specific project needs.

- *A project-directed parts group* performs domain analyses in conjunction with multiple projects. The results are likely to support other projects in addition to the specific projects.

A fourth organizational strategy is also possible, in which a domain analysis is performed as part of a single system development. This is an improvement over the project spin-off, as a domain analysis is performed at the start of the development instead of a post-mortem repackaging. The drawbacks are that the analysis is not likely to cover as many specific needs as a multi-project effort, and the results are not as likely to be generic as is possible with an autonomous effort.

SofTech recommends that a domain analysis effort be within the scope of a single organization or project office. "No one else can provide adequate access to information on the requirements of the applications or evaluate the usefulness of the results." The sponsoring organization should provide the funding, monitor the development, resolve any issues, support its use on applications, and receive the benefits of reuse. [QUA88]

2.1.5 Role of Existing Systems

Analysis of the surveyed reports yielded three major strategies for performing a domain analysis, which differ based on the extent to which existing systems are used in defining the domain. They are:

- o *Commonality analysis*, where multiple existing systems are analyzed to derive a common domain model
- o *Generalization approach*, where one existing system is generalized to derive a reusable domain model
- o *Custom development*, where a reusable domain model is developed from scratch

For Ada developers, commonality analysis seems to be the most prevalent strategy for performing a domain analysis, possibly due to the visibility of the CAMP effort.

The CAMP domain analysis incorporated a commonality analysis of ten existing in the missile systems domain. [MCN86a] The Software Engineering Institute is continuing development and refinement of the CAMP approach [SEI88b], and the Software Productivity Consortium also has a project which is generalizing the CAMP work and mapping the approach into its own framework [PYS88]. Computer Technology Associates also elected to perform a domain commonality study to develop a common model of ground control software for NASA space missions. [MOO88]

General Dynamics (DARTS system) promotes generalization as a satisfactory approach to domain modeling. With this approach, an existing system is modified to become more general-purpose, or reusable. The resultant model is called a *system archetype*. The archetype is then adapted to build a new system according to the needs of each application system.

The concept behind generalization is somewhat different from a commonality analysis. The goal of a commonality analysis is to come up with a solution that is *representative* of enough systems to ensure reusability across a broad range of applications. The goal of generalization is to come up with one solution that is hopefully *sufficiently* reusable.

2.2 Process Models for Domain Analysis

The investigation of process models for domain analysis addressed the following questions:

- o What are the activities involved in the domain analysis process?
- o What are the relationships between the domain analysis activities (that is, what are the major control and data flows)?

The surveyed reports identified dozens of activities which could be associated with a domain analysis. An analysis yielded the following major activities for accomplishing the process of domain analysis:

- o Identify the Need for Domain Analysis
- o Define the Strategy for Domain Analysis
- o Identify the Boundaries of the Domain
- o Select and Acquire Domain Analysis Resources
- o Define the Domain
- o Define Generic Architecture(s)
- o Develop Reusable Components
- o Verify and Validate the Domain
- o Maintain the Domain

Not all of the approaches surveyed included all of these major activities as part of domain analysis. For example, Arango considers the development of reusable components (he calls them "reusable encapsulations") outside the scope of domain analysis. [ARA87] Others do not distinguish the domain objects from the reusable components, and the activity for defining the domain is merged with the activity for defining the reusable components. Defining a domain analysis strategy, and maintaining the domain are activities overlooked by most of the approaches surveyed.

Occasionally, the approaches surveyed differed in the ordering of the major steps to be performed. For example, SofTech recommends defining the strategy for software reuse after defining the application problem domain and analyzing its requirements. [QUA88] Moore recommends constructing a strawman domain definition prior to an examination of existing systems. [MOO88]

While some approaches only describe a sequential application of these activities, parallelism and iteration are explicitly discussed by many of the efforts.

The following sections address each of these activities in more detail, including identification of substeps that may be performed.

2.2.1 Define the Strategy for Domain Analysis

Prior to embarking on a domain analysis, a number of preparatory steps should be performed. Prieto-Diaz calls these preparatory steps *pre-DA* activities. *Pre-DA* activities include "defining and scoping the domain, identifying sources of knowledge and information about the domain, and defining the approach to DA." [PRI87]

At this stage, scoping of the domain involves establishing the general domain subject area(s) to be analyzed. Neighbors is one of several researchers who proposed that the "strategic planning arm of the organization determines the problem domains of interest." [NEI81]

Criteria for selecting the initial domain(s) are rather subjective. SofTech found that "[t]here are no hard and fast rules that govern the definition of a domain, but rather several things that should be considered." [QUA88] These considerations include the following:

- o Similarity of applications in the domain. "Enough" similarity indicates a "reasonable" potential for sharing components.
- o Commonality of hardware and software environments in the domain. The more similar the environments, the easier it is to develop components that are reusable in that domain.
- o Number of applications in the domain. "Enough" applications are required to offset the cost of developing reusable components. The number of applications required to satisfy this criteria increases as the differences among applications in the domain increases.
- o Relationship of the domain to development organizations. The domain should be within the scope of a single sponsoring organization.
- o Size and maturity of the domain. Applications too small or temporary to benefit from the reuse of components should be eliminated.

Additional criteria that may be used in a cost/benefit tradeoff analysis is the availability of resources to perform the analysis and comparative criticality of the domain(s) under consideration. Further refinement of the scope of the domain analysis will occur as the domain analysis progresses.

The approach to reuse and an overall domain analysis strategy is also selected at this time. An analysis of the domain's requirements may be needed to determine the appropriate strategy. [QUA88] Compatibility with other reuse efforts may also be a consideration.

Once the overall domain analysis strategy has been identified, the methodology to be used should be defined in detail. The methodology should address the principles, methods, representations, and tools for performing a domain analysis. [ARA87] Prieto-Diaz proposes the development of a generic set of domain analysis guidelines which are then customized to create a domain-specific document for each domain to be analyzed. [PRI87]

Source(s) of domain information, engineering and support personnel, and automated tools which are needed to perform the domain analysis should also be identified. A set of appropriate resources will depend on the domain(s) to be analyzed and the methodology to be used. Arango notes that the lack of resources may lead to compromise in the methods and representations used for the domain analysis. [ARA87]

2.2.2 Identify the Boundaries of the Domain

According to Iscoe, "[m]ost real-world domains are neither well defined nor clearly bounded." [ISC88] Almost all of the case histories surveyed described problems in identifying and setting the boundaries of the domain(s). *Complexity* was the primary source of trouble.

The CAMP researchers reported on three factors which introduce complexity into the domain identification process: fuzzy domain boundaries, domain overlaps, and domain intersections. [MCN86a]

Domain boundaries are called "fuzzy" when there is not a clear definition of the capabilities which should be provided by systems in the domain. "What one person might firmly believe to be within a domain another person would not." [MCN86a] CAMP recommended that areas of contention be identified early in the domain analysis, particularly if resources for performing the analysis are limited.

Overlapping domains refers to the situation when some areas in an application domain of interest also fall within another domain. Moore cites an example of a facility that is an essential part of one domain, but which "has its own set of typical data structures and algorithms, and is typically developed by a dedicated ... organization." As such, they had valid reasons for including and excluding the facility within the domain. A pragmatic solution was chosen. The facility was defined to be a "distinct (although related) domain," which would reduce the complexity of the original domain. [MOO88] CAMP noted that since the results can be shared overlaps can reduce effort, but warned against getting sidetracked into analysis of the overlapped domain. [MCN86a]

Domain intersection refers to the situation when functional areas within an application domain of interest apply to a wide spectrum of domains. CAMP termed the former *horizontal domains* and the latter *vertical domains*. When resources are limited, sidetracking was also seen as the primary danger of domain intersections. Analysis of the functional domain may expand across the range of application domains at the expense of the original domain. Nise and Giffen called these two kinds of domains *functionally-oriented* and *application-oriented* domains. They recommended investigating possibilities for expanding the *domain of applicability* of functionally-oriented components as early in the development process as possible. [NIS86]

Neighbors reports on two related, but slightly different, limits which must be established for the domain analysis: the *depth of analysis* and the *width of analysis*. The depth of analysis problem determines "how much support should be put into the library domain itself and how much should be required of the supporting domains?" The width of analysis problem involves "a trade-off between specialization and generality," determining "is this function required by most of the systems built in this problem domain?" [NEI88]

The task of cost benefit analysis is made easier by bounding the domain. "It is possible to make reasonable estimates of the number of applications that would benefit and the contribution that would be made to each application." [QUA88]

Domain bounding is not only for the purpose of restricting or assessing the domain development effort. Arango also applies the notion of domain bounding to restricting the search space of a reuser, making the task tractable and more efficient. [ARA87] Domain identification is closely related to domain classification activities, since classification also involves determining which domain(s) an object "belongs" to.

Lubars recommends a different orientation to the bounding problem. He recommends performing more separate, smaller domain analyses. An analysis across these multiple domains is then performed to derive larger, more abstract domains. [LUB88] As such, the iterative process of domain analysis results in an increasing, rather than restrictive, domain scope. A tradeoff analysis must be made between the expense of performing each abstraction against the subsequent effort to discover components (presumably, it is easier to find suitable components as they become more abstract). Lubars provides several specific heuristics to help decide whether multiple domains should be abstracted into a single domain. Three heuristics include [LUB88]:

- o The fraction of similar components is high (greater than 80%) relative to the total number of components.
- o The components differ in terms of only a few properties. He suggests an 80-20 rule, where 80% of their differences could be expressed in terms of 20% of their properties.
- o The potential for reuse of design schemas and other domain knowledge is high.

A related issue is scoping the capabilities of components in a domain. For example, a given component might need a certain set of capabilities when supporting one domain, but needs a few extra capabilities when supporting a second domain; however, a few less may be needed when supporting a third domain. The problem of *domain conflicts* must be addressed if the analyst attempts to support multiple domains in defining a component. Robinson describes how domain conflicts can even occur within a single domain when considered from multiple perspectives or "selfish views". Techniques for detecting and resolving such conflicts must be established as part of the domain analysis methodology. [ROB88]

2.2.3 Select and Acquire Domain Analysis Resources

At this time, specific resources required to perform the domain analysis should be identified, evaluated and selected. As noted earlier, three kinds of resources should be addressed:

- o Knowledge about the domain and the domain analysis process
- o Personnel and organizations for performing the process
- o Tools and environments for automating the process

The primary sources of domain knowledge, recommended by the surveyed reports, are domain "experts" and documentation from existing systems in the domain. CAMP identified *domain representation*, the "selection of a set of applications which characterize the domain under investigation", as a major activity in their domain analysis process. [MCN86a] *Availability* and *quality* of domain documentation were cited as primary contributors to the success of a domain analysis effort. Section 2.3 addresses types and sources of domain information and process guidelines in greater detail.

Most of the surveyed reports indicated that the domain information should be collected prior to performing any analysis work. Neighbors recommends *correlating* all the available information and making a determination as to it *sufficiency* before progressing with the domain analysis. [NEI81] According to Barstow, however, certain domain-specific knowledge can be collected as part of developing the domain. [BAR85]

In addition to domain experts, personnel with expertise in performing the domain analysis must be found. According to Arango, such personnel are "the most expensive items in the ... budget," and one of the reasons domain analysis is a costly process. [ARA87] Expertise in the domain being analyzed should not overshadow the need for expertise in general *software engineering* techniques. CAMP reports that both kinds of expertise are required to perform the analysis. [MCN86a] Section 2.6 addresses the personnel expertise needed to support a domain analysis effort.

None of the reports mentioned the *availability* of personnel with domain analysis expertise as a problem, despite the fact that the technology is very immature. It is likely that domain analysis training programs will have to be instituted to develop such expertise within an organization.

As might be expected, automation of the domain analysis process was recommended by reports which considered the issue. In fact, many of the reports mentioned efforts to incorporate domain-specific tools and information into their development and/or reuse environments. However, none of the surveyed reports really addressed the evaluation and selection of tools as part of the domain analysis process. Section 2.7 addresses the tools and environments which have been developed or modified to support domain analysis activities, and identifies some criteria which could be used in evaluating them.

2.2.4 Develop the Domain Model

Domain modeling is the key activity in the domain analysis process (some even say it is the only activity). It results in the specification of a common model characterizing application systems within the domain. Schemes for classifying reusable components are generally tightly bound to these domain models.

Most of the reports surveyed recommended *object-oriented* methods and models for describing a domain. As such, this section generally describes domain modeling activities in object-oriented terms.

The domain modeling process can be broken down into the following sub-activities:

- *Identification of domain objects*. This step involves an analysis of the domain to identify features common to systems in the domain, and to select objects which model those features. A more specific bounding of the domain may also be defined.

- *Specification of domain objects*. This step involves formalizing and documenting the domain objects, perhaps including the definition of a domain-specific language.

- *Classification of domain objects*. This step involves the definition of a domain-oriented classification scheme, and the classification of the objects according to the scheme.

- *Storage of domain objects*. This step involves addition of the domain model to a reuse library or other storage and retrieval system.

These activities are highly interrelated, and most of the reported approaches partitioned and combined them in different ways. The artificial separation of activities in this list does not imply that they can or should be performed as separate steps. For example, separating the identification and specification of domain objects is like separating

the design of a system from the documentation of the system. Each of these activities and their interrelationships are described in the following sections.

2.2.4.1 Identification of Domain Objects

Information about capabilities which are common to systems in the domain is extracted from the domain knowledge sources and is expressed in terms of the domain modeling elements (e.g., objects, operations, and relations.

According to many of the researchers, domain objects should be identified by performing a *commonality analysis* on existing systems in the domain. CAMP found the process "very similar to the analysis performed in the construction of an expert system. Both these analyses are attempts to formalize a body of knowledge... In the case of a reusability domain analysis the knowledge is needed to [distill] abstract requirements from concrete instances of requirements." [MCN86a]

In performing a commonality study for NASA, Bailin encountered problems in dealing with a mix of different methods and documentation standards in analyzing a set of existing systems. He proposed steps where all domain information be *mapped* to a single, common representation to simplify the identification of similar domain objects. [BAI88] If the original systems are not object-oriented, then objects will need to be derived. Moore found that "for the most part", objects can be inferred from non-object-oriented models. However, he found it necessary in some cases to create new objects to impose an object-oriented structure on certain system commonalities. [MOO88]

In general, multiple methods and representations were suggested to address different views or perspectives of the domain being analyzed. This is independent of whether existing systems are used to derive the domain model or the model is developed from scratch.

Based on the bounding activity described in section 2.2.2, some subset of the domain model may be excluded from further development. For example, it may be determined that certain objects are not common to "enough" systems to justify the cost of development. In another case, certain objects may be identified as belonging to a "different" domain outside the scope of the development.

2.2.4.2 Specification of Domain Objects

Once a set of domain objects has been selected for further development, the next step is to formalize and document them. This step may also involve additional analysis to define more complete and/or appropriate abstractions for the objects.

Moore notes that this is an iterative process and the effort involved will probably depend on the "goodness" of the commonality analysis. For their first cut at defining domain objects, "goodness" of the abstractions was not an issue, and the focus was primarily on mapping the domain information into the object-oriented model. [MOO88] As such, the domain identification and specification tasks may be difficult to separate.

Moore's "development experiences suggest that the most tricky aspect of defining objects is how rich to make them. The question is whether to implement a 'spanning set' or a 'rich set' of functions and data for an object. A 'spanning set' is a minimal set of functions and data which can be combined to achieve all required operations on (or by) the object. A 'rich set' contains a spanning set plus additional operations and/or data typically needed by users of the object." Moore goes on to say that this is a trial and error process, and that considerable engineering judgment is required. He suggests that coupling

heuristics may "suggest a layering of object abstractions so that both richness and orthogonality are achieved." [MOO88]

The language(s) used for expressing the domain specification is/are generally selected prior to performing this task, but are sometimes evolved as the domain specification is developed. The Draco paradigm requires the definition of a formal syntax and semantics which is used in the construction of domain-specific language parsers and pretty-printers. [NEI81] The various alternatives for domain specification languages are discussed in section 2.5.2.

2.2.4.3 Classification of Domain Objects

Another step in the domain definition process is classification. According to Prieto-Diaz, "classification is the product of integrating the knowledge inherent to the objects of the collection with the knowledge of the domain where the collection is going to be used." Using his approach, the definition of a classification scheme and the classification of objects according to that scheme are intimately tied to the object identification process and the specification language used to define those objects. [PRI87]

Auer and Adams recommend a bottom-up approach for development of the classification scheme. Initially, each objects is classified according to a single, very specific category. These specific categories are then combined into "classes and hierarchies". The authors did express the belief that a top-down approach could be used as well. [AUE88] Additional information on alternative classification schemes is provided in section 2.5.3.

2.2.4.4 Storage of Domain Objects

Generally, the surveyed domain analysis approaches used various documents as the sole repository for domain information. However, several of the surveyed approaches provided for storage of the domain information in automated domain libraries or some other database.

2.2.4.5 Impact of Levels of Domain Abstraction

Several sources identify multiple "levels" of domain abstraction which must be considered in defining the domain model. The concept is basically the same among all of the proposals: the top levels of abstraction are more application-specific and conceptual, and the bottom levels of abstraction are more implementation-specific and concrete. 'Code' was considered by most to represent the lowest level of domain abstraction, but some considered domain modeling to go all the way down to the target hardware level. STARS identified the need for some undefined level above the conceptual, this level being called "environment." [STA85]

The level of domain abstraction at which the analyst is working can affect the languages, classification schemes, tools, and kinds of products which are used or developed. For example, at the highest levels of domain abstraction, natural language and a simple data dictionary of application-specific terms may be used to document and classify domain objects. At the lowest levels of domain abstraction, programming languages and complex semantic nets may be used to document and classify reusable encapsulations. [PRI87]

2.2.4.6 Impact of Kind of Application

The kind of application (that is, the application domain problem area) can influence the problems analysts may face in defining a domain model.

Barstow found that dealing with large amounts of domain knowledge is the major problem within his domain of expertise, such as oil well logging which is a process control application which includes the acquisition and manipulation of large amounts of data. He claims that analysis of mathematical domains requires a person with "creative insight". Barstow also identified component interaction as a potential problem for some (unspecified) domains. [BAR85]

The CAMP effort identifies domain-specific characteristics of missile systems software which had "an effect on the detailed domain analysis." The CAMP detailed domain analysis addresses the identification and specification of reusable parts (components). The parts were implemented in the Ada programming language. The domain characteristics and the use of Ada had an impact on what parts were designed, how they were designed, and how they fit together. [MCN86a]

Gargaro identifies the following areas as critical considerations for reuse in the real-time systems domain: distributed execution, scheduling, error recovery, and resource control. The methods by which these areas are handled within a domain can influence whether problems arise in "combining reusable parts in an execution environment that is different from those used to originally develop the parts." [GAR88] The lack of formal models of timing, much less a common model of timing, is probably the greatest inhibitor to the reuse of real-time software.

The kind of domain and the size of the domain can also influence component classification activities. Prieto-Diaz proposes that "[s]election of a particular representation structure [for a domain taxonomy] would depend on the kind of domain analyzed. Different forms could be used within the same domain depending on its size." [PRI87]

2.2.5 Develop Generic Architecture(s)

A generic architecture provides a common high level design for systems/software in a domain. It defines the structure and interaction of the reusable encapsulations that implement the domain model. It may be a more detailed definition of the domain model previously developed, but is probably as different from the domain model as software requirements are different from software designs.

In general, multiple generic architectures are possible within the same domain. The number of generic architectures to be developed may be bounded on the basis of a cost/benefit analysis. The architectures which are developed should be evaluated to determine whether to proceed with the implementation of associated reusable encapsulations.

According to SofTech, the use of generic architectures streamlines the development of new applications in a domain. For example, preliminary design for the new application need only be an extension or customization of the generic architecture. [QUA88]

2.2.6 Develop Reusable Components

During this step the domain definition is projected or mapped to the form which will be reused. With approaches such as Draco's, this step addresses the transformation from

domain components at one level of abstraction to components at the next lower level. With approaches such as SofTech's, this step addresses the development of components implementing the generic architecture.

Arango considers the encapsulation task to be outside the scope of domain analysis. [ARA87] Prieto-Diaz also allocates the encapsulation process to *post-DA* activities. [PRI87] CAMP considers the identification of parts to be part of a *detailed domain analysis* phase. [MCN86a] The Reuse Tools and Environments Working Group believes the development of Ada packages as reusable components merely indicates a choice of representation (language) as part of the process of mapping a domain to a lower level of domain abstraction, but still encompasses a domain analysis. [RTE88]

For this document, we will take the interpretation that the mapping/encapsulation process is within the scope of domain analysis.

Arango identifies many "diseases" which afflict the process of mapping from high-level domain models to low-level component encapsulations [ARA87]:

- o Difficulty understanding and managing the propagation of change, hidden assumptions or coupling between the semantics of different encapsulations.
- o Reusability attributes spread among several encapsulations to compensate for inadequacies in the encapsulation language.
- o Encapsulation as a definite semantics-preserving transformation, where often the encapsulation language cannot express certain semantic information (e.g., timing information), or the abstraction's reuse properties are adversely affected (e.g., polymorphism).

Arango concludes that the encapsulation level is not the proper level for analysis and evolution of an abstraction's reusability properties. He likens the process of manipulating reusable encapsulations to manipulating code in a conventional maintenance situation. [ARA87] On the positive side, Arango states that working with the abstractions produced as a result of a domain analysis can greatly improve the performance of the design recovery process in maintaining and porting software. [ARA]

2.2.7 Verify and Validate the Domain

Many of the reports mentioned that domain models need to be verified and validated. The most common approach was to apply the results of the domain analysis to an actual or prototype system development. The size of the domain may influence the approach. Neighbors suggests "hand testing" the domain [NEI88] although this approach may not be feasible for large application domains. Quanrud describes an extremely limited form of this approach where the prototype applications would be subsets of those identified in the original domain analysis. The only application-specific requirements allowed for the prototypes are those testing pre-defined mechanisms for adapting the design. [QUA88]

Instead of building a new system, the domain definition could simply be validated against existing system definitions. Neighbors reports on a domain analysis experiment where the domain model and an example system specification were independently developed and then compared. During the comparison, the domain model identified an error in the system specification. [NEI88] Errors in the domain definition could also be identified using this approach.

Arango argues for formalizing the domain verification and validation process. He defines a model where the domain definition can be evaluated for two notions of completeness: *relative completeness* and *adequate completeness* (absolute completeness is not possible to define). [ARA87]

Relative completeness states that given a set of problems P and a model R of the reuse task, a domain model is said to be complete relative to P and R if for all problems similar to problems in P, a reuser driven by R can derive at least one satisfying solution. [ARA87] In other words, the domain model is relatively complete if it can support the specification of any application system covered by the domain model.

Adequate completeness says that given a pattern of reuse P (a collection of problems), a domain model is said to be complete with respect to P if the cost $R_f(P)$ over a given period of time is less than the cost of restructuring the domain model. [ARA87] In other words, the domain model is adequately complete if the cost of reusing it "as is" is cheaper than the cost of redefining the domain to make it "better". It is important to note that Arango includes a time factor in the definition. The cost of a domain development can be amortized over multiple subsequent system developments.

Arango also proposes an evaluation of the *competence and performance of the reuser* to derive a measure of goodness of a domain model. [ARA87] This evaluation is a domain evolution task in his model and is addressed in section 2.2.8 on domain maintenance.

2.2.8 Maintain the Domain

Domain maintenance addresses the collection and application of feedback on the use of the domain model to build components, and/or the reusability of the resultant components, to improve the domain. Experiences with using related domain models could also trigger a reevaluation of a domain. This activity also addresses the collection and application of information about forthcoming requirements for applications in the area represented by the domain.

Arango proposes that an evaluation of the competence and performance of the reuser be performed to derive a measure of "goodness" of a domain model. Recall that in Arango's approach, the output of the reuse process is an application system built using reusable components under the control of the domain model and some model of the reuse task. Assume the following two additional outputs are kept from the reuse process: "1) reuse histories of successful completions, and 2) in the case of failures, a description of the 'unknown' problem specifications." Histories of reuse successes can be used to learn and compose statistics about reuse patterns which can be used to tune reuser performance. Unknown problems indicate relative incompleteness of the domain model. One approach for dealing with this problem is *demand-driven* as problems occur. An alternative is a *model-driven* analysis of the domain model to look for opportunities to enhance reuse competence through domain restructuring. [ARA87] Both Arango [ARA87] and Barstow [BAR85] recommend maintaining a history of changes to the domain.

Arango concedes that his *model of adaptive reuse* represents "an ideal state of affairs. Continuous analysis and adaptation of a [domain model] is an expensive process, in particular because it involves the domain engineer and domain experts, the most expensive items in the ... budget." [ARA87]

Maintenance of the domain must deal with both planned and unplanned changes in the domain. The requirements for military software systems are constantly changing. Changes in one part of a domain model, generic architecture, and/or set of reusable

encapsulations will have an impact on the rest of the model, architecture and/or set of encapsulations.

The Reuse Tools and Environments Working Group concluded that change *impact analysis* -- the assessment and management of change -- is a critical domain analysis task. Many decisions are involved when changes are identified for a domain, with the most important consideration being existing users of the domain. If the change has minimal impact on users, the current domain definition could be modified. If the change is compatible with the existing domain definition, a new variant of the domain could be created. If the change is not compatible with the existing domain definition, then a new domain may need to be created. The working group recommended that any domain analysis effort give serious consideration to potential future requirements of the systems it is intended to support. Performing a domain analysis based only on existing systems will not accomplish this goal. [RTE88]

Auer focuses on the importance of domain analysis to reduce the impact of change. "[T]here is almost never a problem whose domain definition does not change once the initial domain model has been created... Therefore, when defining a domain model, we should assume that the definition of the domain will change and take precautions to ensure that changes will minimally impact the model." [AUE88]

Auer's concern seems to be confirmed by NASA's experiences in developing reusable component baselines. One baseline has failed to remain usable; the second is failing, and a third is under development). CTA believes the reason for failure is that the developers could not foresee changes that took place in areas such as platforms, data networks, and distributed operations. [CTA88]

In summary, reducing the need for change, and proper handling of the impacts of change must be addressed within the domain analysis process.

2.3 Inputs to Domain Analysis Process

Prior to and/or during a domain analysis, information will be required from external sources distinguished from data flows occurring internally to each of the domain analysis activities. Each report was surveyed with respect to the following:

- o What information is required to perform the domain analysis activities?
- o Where can the information be found?
- o What qualities must this information possess?
- o What are the impacts of availability and quality of the information?

Two classes of required inputs were identified from the surveyed material: knowledge of the *domain*, and knowledge of the domain analysis *process*.

Knowledge of the domain is information defining the application domain itself. Domain knowledge as a domain analysis input is discussed in section 2.3.1.

Knowledge of the process is information about how to perform a domain analysis and its relationship to the reuse process. Knowledge of the process as a domain analysis input is discussed in section 2.3.2.

2.3.1 Knowledge of the Domain

A variety of sources for domain information were identified during the survey:

- o Existing application systems or prototypes
- o Prior domain analyses, including similar or lower-level domains
- o Textbooks, periodicals and journals
- o Potential "customers" of the domain
- o "Experts" in the domain
- o Requirements for future systems in the domain

Using existing systems as a source of domain information was proposed by almost every approach surveyed. The following different kinds of existing system documentation were used in performing the CAMP commonality analysis: requirements specifications, design specifications, and code (as a last resort). [MCN86a]

Existing application systems serve as examples of specific instances of domains. They are an excellent source of domain knowledge for constructing a domain specification, particularly of what "works". This knowledge is the primary rationale for performing a commonality study as part of domain analysis. The analysis of multiple existing systems also aids in the identification of what can be *different* among systems in the same domain. Existing systems can also be used as "test data" to verify that a domain specification can support the capabilities of actual systems. [NEI88]

The major drawback to the sole use of existing systems in the development of a domain model is that existing systems fail to address the requirements of future systems in the domain; otherwise; there would never be a need to build new systems.

Neighbors proposes the reuse of existing domain objects and operations in the definition of new domain models in the use of Draco. [NEI81]

For more mature domains, information may also be available in textbooks or journals. This information is most likely used for broadly applicable domains such as language processing, encryption, or database management. CAMP suggests textbooks as a potential source for pattern components. [MCN86a]

Experts who have built many systems in the domain are a source of valuable information often not documented with existing systems. The Reuse Tools and Environments Working Group emphasize the importance of addressing the requirements of future systems that might be built in the domain instead of only looking at existing systems. [RTE88]

2.3.2 Knowledge of the Process

Knowledge about the domain analysis process is generally documented in a domain analysis guidelines document, such as section 3 of this document or can be acquired from a methodology "expert". If details of the procedures for domain analysis can be captured and automated, it can reduce the need for domain analysis experts during application development.

2.3.3 Impact of Information Availability and Quality

The availability of domain information, how representative the information is of the domain under consideration, and the general quality of the information are all critical characteristics which should be considered with respect to the domain analysis process. [MCN86a]

The availability of domain information can impact the approach used for domain analysis. If many existing application systems are available, it is likely that a suitably representative set can be found for performing a CAMP-style commonality analysis. If few existing systems are available, a generalization approach may yield better results. Existing systems from related domains may yield portions which are useful. If no existing systems are available or suitable, then the domain must be constructed from scratch.

SofTech claims that in any given domain, it isn't unusual to find some existing applications that are somewhat representative of the domain as a whole. [QUA88] The CAMP developers found many existing missile systems, but carefully selected ten which they felt were suitably representative of their domain. They found that time and personnel constraints are most likely to affect the number of applications that can be studied. [MCN86a] Arango cites the DoD Reusability Guidelines [DOD86] as well as CAMP as being representative of approaches driven by economic and managerial concerns. [ARA87]

Prototyping a domain specification from scratch was a successful approach used by domain experts at a recent domain analysis workshop largely due to the availability of multiple "experts" in the domain. [NEI88]

2.4 Methods For Domain Analysis Activities

This section addresses the specific methods or techniques which were recommended for performing the various domain analysis activities. The following questions were applied to each of the reports:

- o What methods are used for accomplishing each of the domain analysis activities?
- o Can alternative or multiple methods be used? What are the criteria for selecting a method?

Not much information was provided on specific domain analysis methods used or recommended by the various efforts. Because of the lack of detail provided by the efforts surveyed, this section currently provides a kind of "shopping list" of methods. Methods for the following subset of domain analysis activities have been proposed or used by the surveyed efforts:

- o Domain analysis methods in general
- o Methods for domain bounding
- o Methods for identifying objects (part of the domain definition activity)
- o Methods for commonality analyses (part of the domain definition activity)
- o Methods for classifying objects (part of the domain definition activity)

o Methods for developing reusable components

o Methods for maintaining domains

The specific method recommendations for each of these areas are addressed in the following sections.

2.4.1 Domain Analysis Methods in General

Arango's report was the only one which explicitly addressed requirements for domain analysis methods. The following two requirements were imposed. First, the methods must be systematic, domain-independent, encapsulation-independent. Second, the methods must support documentation of the evolution of the domain model. [ARA87]

Arango states that specific domain analysis methods will vary depending on the domain model, the model of the reuse task, and the model for competence and performance of the reuser. For example, methods effective on models for software construction may not be appropriate on models for specification analysis. [ARA87] Prieto-Diaz documents the need for customizing methods based on the application domain being analyzed. [PRI87]

2.4.2 Methods for Bounding Domains

CAMP relied heavily of the use of management control and reviews to ensure that all aspects of the domain analysis remained limited to the domain of interest. [MCN86a]

Many researchers hinted at the use of various cost/benefit tradeoff analyses that might be performed. Moore reports that reduction of domain complexity is a factor in allocating capabilities to particular domains. [MOO88]

2.4.3 Methods for Identifying Objects

The following paragraphs present some findings which apply to the identification of domain objects and some methods for performing a domain commonality analysis.

With respect to identifying domain objects, Neighbors reports a number of modeling techniques that were successful. These techniques include data flow modeling, entity-relationship modeling, semantic nets, and object-oriented designs with and without inheritance. He identifies a minimal set of techniques that should be applied to any domain analysis effort: data flow modeling, control flow modeling, and the modeling of object hierarchies. [NEI88]

Bailin recommends any or all of the following techniques for performing a domain analysis: entity-relationship modeling, data flow modeling and object modeling. [BAI88]

Arango suggests two methods for acquiring domain abstractions (identifying objects). One is a demand-driven acquisition of knowledge from domain experts, and the other is a model-driven restructuring of the domain model. He favors a model-driven method "to maintain the results as independent as possible from individual differences among domain engineers. He further requires that the method for acquiring domain abstractions support incremental acquisition. [ARA87]

Auer and Adams propose that domain objects be identified in a particular order: 1) identify domain-independent objects, 2) identify functional objects using the independent

objects, 3) identify supervisory objects keeping track of independent objects, 4) build abstractions and support objects as duplicate functionality is found. [AUE88]

Gargaro states that the domain identification process must consider whether one is working in a single processor environment versus a distributed environment. For a distributed environment, he recommends using a pre-partitioning technique and adhering to specific guidelines for composing reusable applications. [GAR88]

2.4.4 Methods for Commonality Analyses

The CAMP researchers discovered multiple levels of commonality and found that these levels have a major impact on the way a domain analysis is performed.

In performing a CAMP-style commonality analysis, the analyst looks for the presence of functional, pattern, and architectural commonalities among existing systems in the domain, as follows [MCN86a]:

- o Functional commonality - the analyst takes a "black box" view of the systems to identify common operations. This is "the type of commonality classically associated with software reusability."
- o Pattern commonality - the analyst looks for recurring patterns of logic in the systems, such as a finite state machine.
- o Architectural commonality - the analyst looks for common models of major components of the systems.

"If a domain analyst only looks at one level, then commonalities at other levels will be overlooked." They identified a tendency of analysts to look only for commonality of operations, a low level of commonality. Pattern commonality was identified as an important type of high level commonality which will lead to the development of schematic parts. [MCN86a]

CAMP recommended the functional strip method for performing a commonality study. Using this method, the analyst examines a particular function across all domains in the set being analyzed. CAMP found that the narrowness of the strips influenced the ease and probability of finding commonalities. They found that narrowing the strips made it easier to recognize similarities, but narrowing the strips too far made it impossible to recognize high level commonalities. They also used data flow modeling to highlight similarities between the functions. [MCN86a]

In performing their commonality analysis, CAMP found that two "barriers which need to be overcome in this analysis are arbitrary differences and notational differences." Both problems proved to be major hindrances to detecting commonality, and required both application and software engineering expertise to resolve. [MCN86a]

2.4.5 Methods for Classifying Objects

Prieto-Diaz's recommended method for creating a domain classification scheme (taxonomy) is to start with the obvious and simple domain classes and develop the taxonomy incrementally as domain knowledge is acquired. [PRI87]

Auer and Adams recommend a bottom-up approach to object classification. Specific object categorizations are created, combined and then generalized into the final hierarchy. [AUE88]

Iscoe recommends using the scaling theory to formalize the mapping between user specifications to hardware implementations. He believes this approach will lead to a methodology usable by a domain expert aware of the "natural" classifications of objects within domains. [ISC88]

2.4.6 Methods for Developing Reusable Components

Prieto-Diaz's recommended method for encapsulation is "similar to that used in object-oriented programming or McCain's object domain analysis." [PRI87] On selecting the objects to encapsulate, Prieto-Diaz recommends selecting those with the most potential for reusability first. [PRI87] Quanrud also recommends this criterion. [QUA88]

The CAMP report identifies three main issues for specifying the components to be implemented in Ada: "what constitutes a part, how the part will be used, and how to specify a part so that it is reusable." Methods and languages for specifying the requirements and design of parts (and their environments) were developed specifically for the CAMP effort. The following six methods were defined for developing reusable Ada parts: typeless, overloaded, generic, state machine, abstract data type, and skeletal code. Methods for accessing the parts were also developed. These methods addressed whether the parts were bundled or unbundled. [MCN86a]

2.4.7 Methods for Maintaining Domains

Arango requires that the methods supporting domain evolution "should proceed *incrementally*, must *guarantee improvements* in the competence and performance of the reuser, and must preserve the *well-formedness* of the [domain model]." [ARA87]

2.5 Product of Domain Analysis Activities

A domain analysis is expected to result in a number of products. The following questions were applied to each of the surveyed efforts:

- o What are the products resulting from each of the domain analysis activities?
- o What languages, data structures, etc. are used to represent the domain analysis products?
- o How much of this information is stored? Where and how is it stored? How is it found and accessed? How is it presented?

Each of the activities in the domain analysis process results in one or more products. Kinds of products identified during the survey are:

- o Domain models
- o Domain languages
- o Domain classification schemes
- o Domain object abstractions
- o Domain development guidelines

o Domain reuse guidelines

o Generic architecture(s)

o Reusable component encapsulations

o Component development guidelines

o Component reuse guidelines

o Domain rationale and history

o Application development environments

The following sections describe each of the possible domain analysis products.

2.5.1 Domain Models

Most of the surveyed approaches recommend object-oriented models for describing a domain. Knowledge-based structures were the second most recommended structure for domain models. The rest did not specify any particular kind of model that should be used. Presumably, a generic architecture approach might utilize whatever model was characteristic or appropriate for the particular domain, although Quanrud also recommends an object-oriented approach. Several of the approaches surveyed recommended using multiple methods and representations for specifying the model.

The set of modeling elements common to all the object-oriented approaches surveyed includes objects and the operations which can be performed on each object.

Several of the surveyed object-oriented approaches explicitly incorporate the concept of inheritance into their domain models. Inheritance provides that new objects can be defined in terms of existing objects. Existing objects might be used "as is", "specialized", or with new capabilities added. Usually, an object can only inherit from one other object, forming a strict hierarchical model. Multiple inheritance provides that objects can inherit elements from more than one other object. Lieberherr and Riel are among some who believe that support for multiple inheritance is essential to domain modeling. [LIE88]

McGregor recommends a "one-to-one correspondence between inheritance structures and domains." [MCG88] Sayrs and Peltier identified the choice of a taxonomic (inheritance hierarchy) or non-taxonomic (any one-to-many relation) object organizations as an "important dichotomy that exists in applying the object approach to one's domain." They found that both can be "combined with advantages in efficiency and design clarity." [SAY88]

Several approaches include and focus on the relationships between objects (also interpreted as the structure of the model). One use of relationships in such domain models is for component classification and searching. Another use of relationships is to define mappings between domains [NEI88]. Other uses of relationships are to define how components are structured into more complex systems [MOO88] or how components interact [GRE88].

Still other approaches include the attributes, sometimes called properties or characteristics, of an object. Lieberherr and Riel focus on this aspect and found factoring common attributes and specialization of attributes to be the most useful aspects of inheritance to apply to domain models. [LIE88] Sharing of object attributes is the criteria

used in forming conceptual clusters of objects. This process of grouping and regrouping objects into conceptual clusters was identified as by Prieto-Diaz as one of the basic activities of domain analysis. [PRI88]

Lubars identifies the specification of constraints on attributes (as part of developing the object hierarchy) as the most difficult engineering task. He further states that the task becomes increasingly more difficult as the level of abstraction is raised. Lubars also recommends including information about the mapping between objects and the attributes that distinguish them in the domain model. [LUB88]

Iscoe's model incorporates domain-specific scales into the definitions of objects and operations. The scaling theory is used "as a means for classifying object attributes in a manner that is both formal and yet captures enough of the underlying real world semantics that he resultant model can be used for specification and transformation purposes." Using the scaling theory also resulted in "a new approach toward solving some of the problems of multiple inheritance." [ISC88]

The Draco paradigm includes the definition of rules of exchange between objects and operations in the domain in terms of language translations. Domain-specific procedures for optimizing the transformations may be defined if they are needed. The transformations are stored in a library separate from the library of components. A component is one object or operation in the domain. [NEI81]

The SofTech report on generic architectures recommends that information about the application domain environment (dependencies on external systems) be included. [QUA88]

Quanrud's report promotes the following advantages of object-oriented approaches [QUA88]:

- o The components tend to be "inherently reusable"
- o Large components are already likely to be object-oriented
- o Object-oriented design (OOD) highlights opportunities for reuse
- o OOD facilitates traceability of design to requirements

Biggerstaff believes object-oriented representations have strong operational capabilities, but also criticizes the representations because of weak informal descriptions. He also believes they are too target-specific. [BIG88] While this fact may be true of object-oriented *programming languages*, his comments in general would probably not be shared by most other researchers with respect to object-oriented representations.

2.5.2 Domain Languages

Arango warns that the choice of representation is fundamental because the representation will limit what domain information can be modeled, what domain information can be analyzed, and what domain information can be learned. He recommends that the representation(s) chosen be machine processable. [ARA87]

Neighbors reports on a variety of representations which have been used for domain modeling. These representations include "data flow diagrams, entity-relationship diagrams, semantic nets, object diagrams, [and] class hierarchies [with] inheritance." He found that capturing multiple 'flavors' of the domain is advantageous, but noted that

"[u]sually only one each from the object hierarchy, data flow and control flow representations need be used to analyze a domain." [NEI88]

All of the software generation approaches surveyed define a domain model using a domain-specific language (which is also input to a transformation tool). Prieto-Diaz also specifies the use of a domain-specific language for defining domain models, but does not prescribe the method for deriving reusable components. [PRI87]

Domain-independent languages (such as Classic-Ada [SPS88], or an entity-relationship definition language) were also recommended by some approaches. Baxter recommends the choice of an algebraic versus an operational language for domain specification. [BAX88]

The languages which were identified by the survey tended to become more domain-independent at lower levels of domain abstraction. However, Barstow defines the specification of domain-dependent information at all levels of abstraction. [BAR85] Biggerstaff recommends four different languages for specifying a given object, ranging in degree of formality and abstraction, independent of domain-specific content. Each language has a different role and set of capabilities. He considers the use of informal abstractions the key property of his domain model, since developing an understanding of a large complex program can be impossible otherwise. [BIG88]

Special languages may be required when dealing with multiple domains. Lieberherr states that "[d]omain models can be combined using techniques from module interconnection languages." [LIE88] The Draco system addresses domain interconnections at different levels of abstraction using a language for specifying transformation rules. [NEI81] Baxter requires that the domain specification language support "quick and easy establishment" of relationships between domains. [BAX88]

The definition of a Draco domain language includes the specification of a parser describing the syntax and internal form of the domain and a pretty-printer describing the external form of the domain as shown to users. [NEI81]

Several efforts recommended the use of both textual and graphical representations for domain information.

2.5.3 Domain Classification Schemes

Prieto-Diaz identifies three basic classification structures: hierarchies, semantic nets, and faceted schemes. He believes that the analysis can use either a single one of these structures to represent the domain taxonomy, or it could use a collection of those structures. Different structures might be used for domains at different levels of abstraction or for different parts of the same large domain. The chosen structures will also depend on the kind of domain being analyzed. The domain taxonomy could be derived from a complete domain definition or could be broken down into partial classifications. [PRI88]

Because it is likely that a large number of parts will be identified by a domain analysis (the CAMP domain analysis resulted in the identification of over 200 parts), CAMP recommended the development of a Software Parts Taxonomy to help domain analysts organize their work. As parts were identified, they were classified according to the taxonomy. The parts were additionally classified according to how they could be implemented: simple, generic or schematic. The parts were furthermore partitioned into two classes according to whether they were domain-dependent (applied only to the domain of interest) or domain-independent (useful in other application domains while still being highly relevant to the domain of interest, such as math parts). [MCN86a]

In some cases, the domain model itself represents a scheme by which reusable encapsulations might be classified. For example, a semantic net representing the interactions between objects in a domain could be used to classify (and show possible relationships between) the encapsulations implementing those objects.

2.5.4 Domain Object Abstractions

Arango requires that the domain abstractions be such that it is "possible to rigorously assess the relevance of acquired abstractions, and the relative completeness of the [domain model]. The representation of reusable abstractions should allow for economic projections onto practical encapsulation languages." [ARA87]

Prieto-Diaz proposes that object abstractions and groups of abstractions be described by frames. "Frames seem to be a natural representation of domain objects and functions and provide the necessary mechanisms to express relationships." [PRI88]

Gargaro believes that adequate documentation of components will be essential to their future reuse. The documentation must minimally include a functional description of the component, timing behavior, runtime environment dependencies, and a description of how it is used within the domain. [GAR88]

CAMP requires that a part specification "inform the part user of what the part accomplishes functionally and also provide performance information about the part. The specifications must be documented in accordance with the Software Requirements Specification (SRS) DID of DOD-STD-2167, and must facilitate communication of the part's characteristics. A final requirement of the specification is to provide an environment for the part, describing the dependencies which existing between parts." [MCN86a]

2.5.5 Domain Development Guidelines

Domain development guidelines describe how to perform a domain analysis. Prieto-Diaz proposes that a generic domain analysis guidelines document be developed first. Domain-specific guidelines documents would be derived from the generic version for each domain being analyzed. He recommends that the domain-specific guidelines document include "a high level breakdown of activities in the domain, which part of the domain to analyze, potential areas to modularize, standard examples of available systems, and any issues relevant to that domain." [PRI87]

2.5.6 Domain Reuse Guidelines

Domain reuse guidelines describe how to reuse the domain definition in a new systems development. They could also address how to maintain the domain definition or how to incorporate the domain definition in an analysis of a new but related domain. Note only reuse guidelines identified during the survey addressed the reuse of the resulting encapsulations, rather than the domain model itself.

2.5.7 Generic Architecture(s)

SofTech proposes the following information be documented about each generic architecture developed [QUA88]:

- o Architecture
- o Control relationships

o Data global to the architecture or component

o Detailed requirements

o Ada package specifications for each component

o Adaptation requirements for each component

o Assessment of the likelihood of requiring adaptation

They found that the top level design produced during the domain analysis should be more complete than for a "normal" development and provide enough detail to determine if the architecture is adequate for use on a given application. [QUA88]

2.5.8 Reusable Component Encapsulations

Reusable encapsulations are usually modules written in some programming language. For purposes of this document, we are constraining them to be Ada program units. Some researchers further constrain them to be Ada library packages (versus subprograms or nested units).

Three kinds of component implementations were identified by the CAMP effort: simple, generic and schematic. A simple part is an Ada package which is used "as is". A generic part is an Ada generic package which is reused through *instantiation*, a process of tailoring the data types and/or subprograms used by a package. A schematic part consists of a template and a set of construction rules; parts are reused through a process similar to software generation. [MCN86a] Schematic parts can be rather complex, and in general include multiple parts. They might best be considered more akin to a generic architecture for a low-level domain.

The primary content of a CAMP part specification includes its functional capabilities (inputs-processing-outputs), performance information, and dependencies with other parts. The part specification provides a domain-oriented definition of the actual part which is implemented in Ada. [MCN86a]

2.5.9 Component Development Guidelines

Component development guidelines address how to develop reusable components. Several general-purpose guidelines, such as the SofTech Ada Reusability Guidelines [BRA85], have been written which address the development of reusable Ada software. The SofTech guidelines also provide guidance in the need for and structure of generic architectures.

Prieto-Diaz calls such guidelines *encapsulation standards*. He recommends that they address how to make specific components reusable by "proper structuring, modularization, and standardization." [PRI87]

CAMP found that characteristics of the application domain will impact what parts are designed, how the parts are designed, and also how the parts fit together. [MCN86a] For example, the efficiency of parts for real-time systems is a critical part design requirement not necessarily imposed on parts for other domains. As such, domain-specific guidelines may be more appropriate in some cases.

2.5.10 Component Reuse Guidelines

Component reuse guidelines describe how to reuse the components which are available for new systems development.

Prieto-Diaz defines two related products: *reusability guidelines* and *domain standards*. The reusability guidelines indicate "how to reuse individual components, list performance considerations, limitations, and potential for reuse (e.g., where can they be reused.)... Domain standards are refinements of the guidelines focusing on standards for building systems in the domain." He recommends the following contents: a listing of reusable components available, definitions of logic structures and guidelines to integrate reusable components into systems, and component interface and interconnection problems. If a domain language is available, then the transformations refining the domain definition into intermediate domains to an operational representation are included. [PRI87]

The CAMP effort resulted in both manual guidelines and automated support for accessing and integrating the reusable parts into an application. Their approach is based on the premise that search and retrieval of parts must be domain-oriented. [MCN86a]

2.5.11 Domain Rationale and History

Domain rationale is information about why particular decisions were made in defining a domain or component.

Neighbors reports the finding that "*domain analysis rationalization* is one of the most important results of the process. This finding was held even by individuals initially having a *constructive point of view*, one which emphasize the ability to construct code from the specification. Understandability of the results, particularly communicating the issues and tradeoffs made during the analysis were viewed as very important. [NEI88]

Robinson suggests that rationale which justifies including and/or excluding components from a domain model and rationale which discusses the compromises and/or resolutions for conflicting components are both very important. [ROB88]

Domain history is a log of the changes that are made to a domain over its lifetime. Barstow recommends that the history provide a record of all decisions leading to each domain change. [BAR85]

Domain history can also refer to information about the history of usage of a domain. Arango specifically recommends collecting performance data about a domain which describes how well the domain is reused. [ARA87]

2.5.12 Application Development Environments

According to McFarland, the product of a domain analysis effort is a new software development environment tuned to the particular domain. Key components of the environment are a domain-specific language for writing new system specifications, a tool for translating specifications written in the language into programs implementing new systems for the domain, and a supporting knowledge base containing domain information. [MCF85] The same can be said of other domain-dependent program transformation systems and approaches. Greenspan describes such an environment as containing a knowledge representation framework, domain-specific tools, and a domain-independent conceptual modeling platform. [GRE88] The CAMP effort included the development of a domain-specific toolset, but did not include it within the scope of the domain analysis effort. [MCN86a]

2.6 Personnel Supporting Domain Analysis

The following questions were applied to the surveyed reports to identify possible requirements for personnel to perform a domain analysis:

- o What kinds of expertise are required to perform the domain analysis activities?
- o What level of effort is required to perform the domain analysis activities?

The survey of domain analysis efforts indicates that specialized personnel will be required to perform many of the activities associated with domain analysis. According to the kind of expertise required, we identified the following five unique *roles* for the personnel performing a domain analysis:

- o Domain analysis methodologist
- o Application domain expert
- o Domain analyst
- o Encapsulation analyst
- o Domain encapsulations expert
- o Domain analysis systems specialist
- o Domain analysis project manager

These roles are closely related to roles found on most system developments, except that specific expertise is required in order to support a domain analysis. For example, a domain analyst is basically a systems analyst whose analysis must address multiple systems in a domain rather than a particular system instance.

The CAMP report did not address specific analysis roles, but did emphasize the need to have both application domain and software engineering expertise to perform the domain analysis. They found that having both kinds of expertise were particularly important for addressing differences among existing application systems. [MCN86a]

Each of the domain analysis roles is described in the following sections.

2.6.1 Domain Analysis Methodologist

The person supporting this role provides "the procedural know-how on [domain analysis]." [PRI87] The domain analysis methodologist is responsible for defining the languages, tools and techniques to be used in performing the domain analysis. These could include organization-wide standards and procedures as well as domain-specific.

This person is generally also responsible for training personnel in the use of the domain analysis methodology, assessing conformance to any applicable standards and procedures, and providing expertise in the area of systems/software engineering.

The reader should note that Prieto-Diaz uses the name "domain analyst" to refer to this role of methodologist. [PRI87] The role of a domain analyst has a completely different definition in this document.

2.6.2 Application Domain Expert

The person supporting this role has expertise in the application domain being analyzed. This expertise is based on direct experience building systems in that domain. Ideally, this person also has expertise in application domains related to the domain of interest in order to address commonalities across multiple domains.

This person minimally acts as a consultant to other domain analysis personnel, assisting in defining model(s) of the domain, finding and analyzing existing systems, anticipating and assessing the impact of future system requirements, etc. This role is sometimes combined with that of the domain analyst.

This person could also act as a consultant to systems development personnel building new application systems in the domain. In this role, the domain expert assists in defining reusable component search criteria, evaluating components for suitability, adapting components, and integrating components into the new application system. [RTE88]

2.6.3 Domain Analyst

This person actually performs the domain analysis and documents the domain model. This person may also be responsible for defining any generic architecture(s) associated with the domain.

Like the domain expert, this person could also act as a consultant to personnel building new application systems in the domain. This person can provide more detailed knowledge related to the domain model, classification scheme, generic architecture, etc.

2.6.4 Encapsulation Analyst

This person has expertise in the methods and languages used to develop reusable encapsulations of the domain model. More specifically, this person knows the best techniques for mapping the domain model into an Ada implementation and making sure that the Ada encapsulations are sufficiently reusable.

The generic architecture or other higher-level parts specifications may be influenced by the use of Ada. For example, CAMP found that the use of Ada tasking led to special process control parts. [MCN86a] As such, this person contributes to the more detailed domain analyses. Further, this person may provide guidance to new system developers when it is time to adapt and integrate specific Ada encapsulations into the application code.

2.6.5 Domain Analysis Systems Specialist

As the domain analysis process becomes more and more automated, personnel with expertise in using specific domain analysis tools will be required. This fact is evidenced by experience with the Draco toolset. [NEI81]

Personnel with expertise in storage and retrieval of components from manual or automated reuse libraries will also be needed.

2.6.6 Domain Analysis Project Manager

Only a few reports addressed the role played by management in the domain analysis process. The CAMP report emphasized the role of management personnel in the review process and in constraining the domain development effort to its defined boundaries. [MCN86a] CTA also emphasized the role of management personnel in reviewing components, and in assessing return on investment against continued development. [CTA87a] SofTech describes the role of management in providing funding for the domain analysis effort, monitoring the development, making cost/benefit tradeoffs, resolving issues, supporting its use on applications, and receiving the benefits of reuse. [QUA88]

Once a domain analysis has been done, management can play a role in monitoring the reuse of the domain analysis products and assessing the need for evolution of those products.

2.7 Tools Supporting Domain Analysis

Each of the surveyed reports was studied to determine:

- o What automated tools are proposed to support the domain analysis activities?

Automation of the domain analysis process was highly recommended in reports which considered the issue. Several reports identified various existing tools which might be applied to the domain analysis process, and also identified various non-existent tools which could potentially be developed. However, as Prieto-Diaz notes, it is "difficult to specify tools for a process we do not understand fully." [PRI88]

A wide range of existing tools could be used to support the specification and analysis of domain models, such as entity-relationship modeling tools, data flow analysis tools, object browsers, parsers, etc. The selection of particular tools by domain analysis researchers was influenced by the domain modeling methods which were being used. We suspect that the converse, selection of domain modeling methods based on the availability of automated tools to support them, was also an unstated consideration.

While not part of a "typical" development environment, expert system building tools also exist which could be used to support the extraction of domain knowledge. It is interesting to note that researchers involved in software generation approaches to reuse have been addressing domain knowledge capture and representation for some time, while researchers involved in software construction approaches are just now discovering its potential uses.

Several of the reports described reuse environments into which such domain analysis tools have been or could be integrated. These are:

- o Automated Missile Parts Engineering Expert, a program construction system specific to the development of missile system applications [MCN86b]
- o Software Reuse Environment, supporting object-oriented requirements analysis and design [MOO88]
- o Development Arts for Real-Time Systems, a program generation system [MCF85]

- o Draco, a domain-oriented program generation system [NEI83]
- o Unnamed GTE reuse library system [PRI87]
- o IDeA, a design environment supporting reuse of designs [LUB88]
- o DESIRE, a design recovery system and reuse library [BIG88]
- o Demeter, "a meta-system for object-oriented programming and knowledge representation" [LIE88]
- o Reusability Library Framework, library management system organized around particular domains [SIM88]
- o Ozym meta-generation system (proposed) [ISC88]

Very few of the reports dealt with the issue of new tools which are needed to support activities unique to domain analysis. Prieto-Diaz recommends that such tools be identified by analyzing the domain analysis process to identify activities which are repetitive, and to focus tool developments in those areas. He identified two such activities: conceptual clustering, and finding relationships between domain entities. [PRI88] The Reuse Tools and Environments Working Group also identified the need for conceptual clustering tools. They also suggested the development of tools to support automatic classification of reusable components. [RTE87]

3. GUIDELINES FOR CONDUCTING A DOMAIN ANALYSIS

The guidelines for conducting a domain analysis are based on the following definition of domain analysis:

> Domain analysis is the systems engineering of a family of systems in an application domain through development and application of a reuse library.

The domain analysis must be focused to address a *family of systems* in a particular application domain. The more definitive these family of systems can be defined, the more focused and precise the domain analysis activity can be.

Domain analysis is, in fact, a *systems engineering* activity. It involves understanding and specifying the requirements for the family of systems (in the form of a *domain model*) The domain analysis activity includes developing a design for the family of systems (in the form of *generic architecture(s)*). It must include in this process all of the tradeoff and analysis techniques that are needed for proper systems engineering of any one of these systems (e.g., performance analysis, reliability analysis).

Central to the concept of domain analysis is, of course, reusability. Thus, the end goal of a domain analysis activity is to develop a reuse library *asset* that will be used in the implementation of system instances in the domain family. These assets in the library will include software components and generators, documentation, interface specifications, test plans, procedures and data. In addition, the domain model and the generic architecture(s) are themselves valuable reusable assets.

The domain analysis guidelines are designed to develop a complete and rigorous domain model and associated generic architecture as a precursor to developing a set of reusable components for repeated application in developing systems in the domain. It is our conclusion that domain analysis, when done right, is a significant undertaking --presumably with a significant benefit.

This section presents guidelines for conducting a domain analysis. The recommended methodology is a synthesis of various techniques that have been applied by current researchers in this field. As such, it is sure to continue to be refined as additional experience is gained in this very specialized field.

The following subsections overview the strategies and paradigms for domain analysis, describe the relationship of the domain analysis to the overall software development process and present the detailed steps of the recommended domain analysis process.

3.1 Strategies and Paradigms for Domain Analysis

Based upon the experiences of a number of researchers (including [BIG88], [LIE88], and [QUA88]), and motivated by the end goal of producing Ada software, *object-oriented* is selected as the underlying paradigm for the modeling and architecting of the domain and for the resulting design and development approach for the software component set (see Figure 3.1-1).

Such a unified object-oriented domain analysis process offers several advantages:

- Provides a single, consistent model that requires no "great mental leap" from analysis to design and increases traceability and maintainability

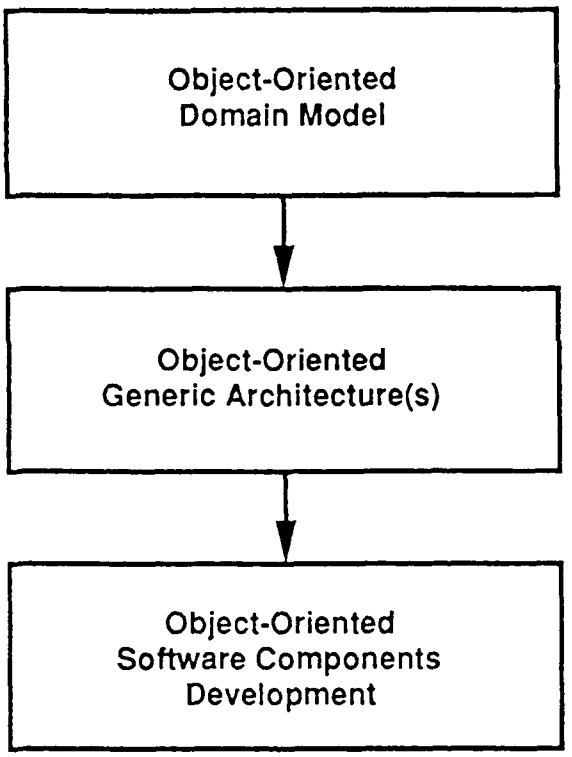


Figure 3.1-1. Object-Oriented Domain Analysis Paradigm

- o Matches the technical representation of the software system more closely to the conceptual view of the application
- o Provides a stable framework for analyzing the problem domain and for levying requirements
- o Supports implementations in Ada using abstract data types

Object-oriented development allows a natural transition from the real-world problem space to analysis to design to implementation. Real-world application entities are specification objects discovered by performing object-oriented analysis. Object-based analysis approaches thus serve to capture a model of reality. Such approaches are well suited to managing the complexity inherent in massive, software-intensive systems.[BOO87]

It is important that the domain analysis be accomplished using a uniform underlying representation model. Multiple representation models are difficult to handle and may be incompatible. For example, components whose design is based upon a domain analysis using a functional decomposition method may not easily be used in an object-oriented system design.

Domain model objects are also architectural design objects. Design elaborates object internals and may discover new objects needed for implementation. Implementation of the object-oriented design is efficiently accomplished in Ada using structured collections of abstract data types. This consistency with respect to the fundamental conceptual units (i.e., objects) across methods provides for a smoother transition between phases, a tighter integration of methods, and a more exact traceability of requirements. [SMI88]

Objects have been observed to be more stable over time than other representations as the foundation for representing domain knowledge: [COA89]

- o Interfaces - highly volatile
- o Functions - very volatile
- o Sequencing - very volatile
- o Data - less volatile
- o Objects - the least volatile

This means that while the characteristics and operations of these objects may evolve over time, the objects that are identified will remain the most stable. For example, decades from now the tactical command and control domain will likely still deal with objects such as threats, weapons, sensors, communication links, geographical locations, etc.

As summarized in Figure 3.1-2, a three phase approach is recommended for the domain analysis:

- o *Model the domain.* Based upon knowledge of the domain and programmatic constraints, an object-oriented domain model is developed and validated. This model is specifically analyzed to denote the necessary adaptations needed by envisioned future systems in the domain

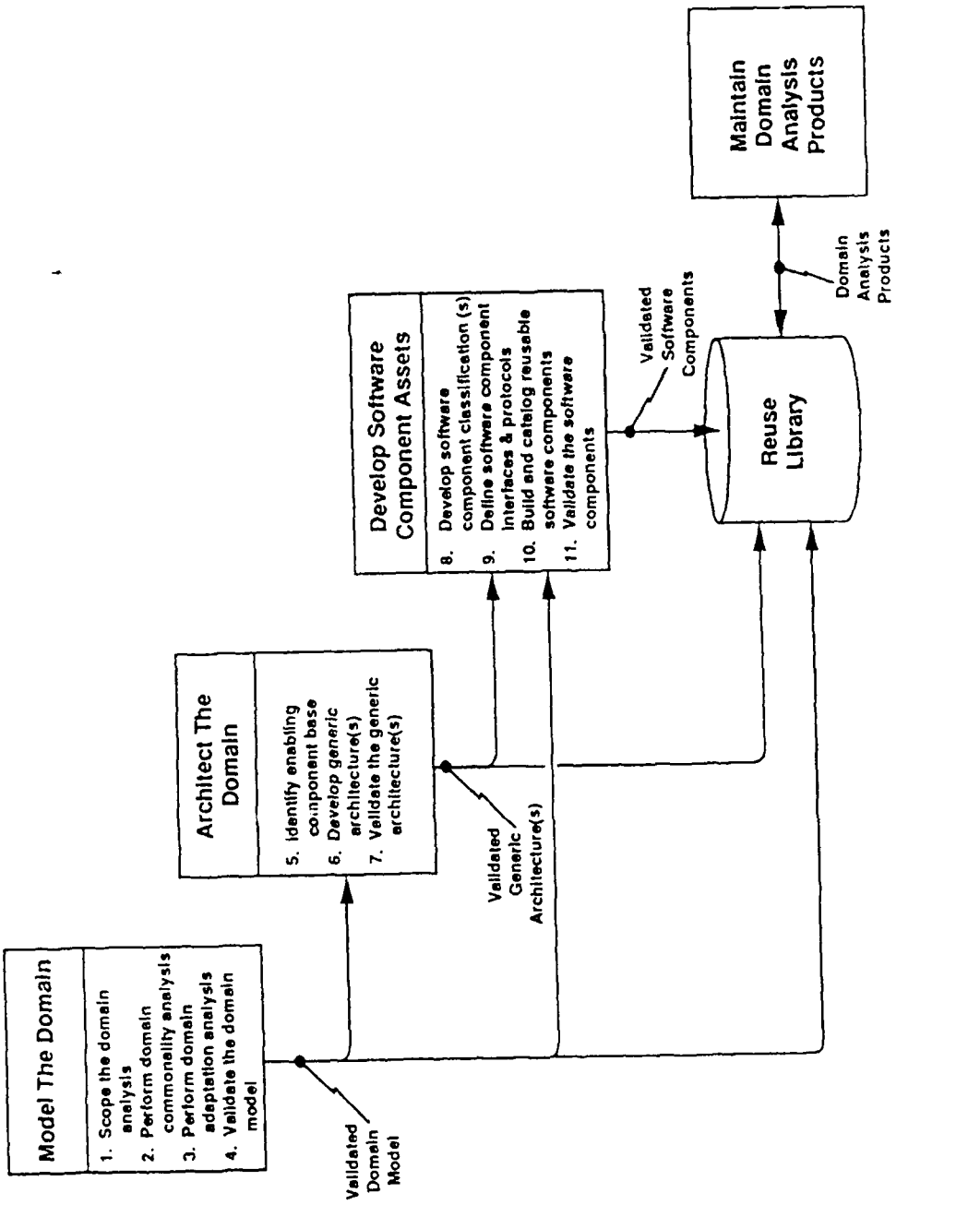


Figure 3.1-2. Domain Analysis Process and Major Products

- *Architect the domain*. A generic object-oriented Ada architecture(s) is developed and validated for the family of systems, identifying those software components that are potentially reusable. Standard systems engineering disciplines are applied to ensure that the architecture is a feasible and satisfactory basis for developing future systems.

- *Develop software component assets*. A set of reusable object-oriented Ada components is built and validated to comply with the interfaces and protocols required by the generic architecture(s). These components are cataloged into the reusable component library for the domain.

The recommended domain analysis activity produces the following set of primary reusable products:

- *A domain model* that defines reusable specification objects which could be included or modified in a rigorous object-oriented specification of a system.

- *A generic architecture(s)* that is reused in each instance of a system in that domain.

- *Software components* and associated classification scheme that are reused in multiple systems in the domain.

Object-oriented methods are recommended for *all* domain analysis activities. An organization performing the recommended domain analysis process should adopt techniques and representations for the various steps in the process that reflect an object-oriented view.

The recommended approach to domain analysis may be applied with techniques other than object-oriented. Such an approach may be adopted for a number of reasons, including:

- Another domain modeling method may be selected because of familiarity with a requirements analysis method and the availability of tools for that method. Transition to an object-oriented Ada design and the associated traceabilities will need to be addressed.

- Different forms of architectural partitioning may be employed to reflect long-standing subsystem boundaries or to take advantage of existing software components. Maintaining consistency and traceability between the domain model and the architecture will require additional attention.

In addition, it is recommended that *all* three steps be taken in performing a domain analysis. Anything less will add risks to the potential payoff of the domain analysis and associated reusable assets. Specifically, it will increase the probability that investments in some reusable assets will be lost because they will be later found not to be suitable for use in developing certain of the systems in the application domain.

Omitting the domain modeling activity and proceeding directly with a generic architecture approach has the following risks:

- Sufficient attention may not be paid to identifying commonality at a high level across the domain because attention was focused toward architectural issues and tradeoffs too soon.

- Detailed analysis may not be accomplished regarding the adaptation requirements of the various domain elements. This analysis provides an understanding of the potential evolution of the application domain. If the architecture does not take these adaptations into account, the architecture and associated component assets may become prematurely obsolete.

- Too much bias may be paid to previous application architectures without due consideration of evolutions in technologies. For example, a functional architecture may be derived that closely matches previous implementations. This architecture and the associated component assets may be obsolete in the context of an Ada object-oriented design. Similarly, an architecture that is appropriate for single processor implementations may be obsolete in a distributed, multi-processor environment.

Similarly, omitting the generic architecture step and proceeding with the development of software components based only upon a requirements level commonality study has the following risks:

- The interfaces developed for the reusable software components may be awkward and difficult to use within an actual architectural context, even though it may be complete and fully functional. Interfaces that seem to be appropriate from the perspective of a component implementor may be difficult from the component user's perspective. This would result in unnecessary application code being written to conform to the component's interfaces, thus lessening the payoff of using the component. In extreme cases, a difficult interface could result in the component not being reused.

- Software components that are developed may not be selected in architectures that are subsequently developed for the systems in the application domain. Architectural decisions may preclude the use of already developed "reusable" assets, either intentionally (as a result of tradeoff analyses) or accidentally (because of lack of knowledge that the component even existed).

- The software components that are developed may not even satisfy a feasible implementation solution, and therefore could not be used in subsequent architectures even where there is a desire to do so. For example, a reusable software component may be developed that meets all of the adaptation requirements of the domain but executes too slowly or takes too much memory to feasibly be used within an actual application context. This problem would not surface until a full architectural analysis is accomplished for the first system instance, typically after the investment in the component had already been made.

- Economic decisions regarding what reusable components to develop may be erroneous. The knowledge of the costs associated with developing the various elements of a system are usually much more accurate after detailed architectural analysis is completed. There is a risk that investments made in some of the components may not have the payoff anticipated or that a high payoff component may not be developed.

Clearly, proceeding with investments in reusable component assets without both domain modeling or domain architecture analyses would incur all of the risks mentioned above.

While the cost of performing all three of the recommended domain analysis steps is likely to be substantial (potentially approaching the cost of a single system implementation), it is our belief that such an investment is necessary to achieve high percentage reusability in an application domain. Further experimentation and demonstration is needed, however, to validate this hypothesis.

It should also be noted that in actual practice the three domain analysis phases will be somewhat iterative, with refinements to previous models or architectures being identified throughout the process. After an initial set of reusable products are developed, there is a maintenance activity to keep the assets current with evolutions of the application domain and in technology. Additional details regarding the steps performed in each of these phases is presented in section 3.3.

(This space intentionally left blank)

3.2 Relationship to the Overall Software Development Process

The three phases of domain analysis are analogous to the three acquisition phases:

- *Model the Domain* develops a complete specification of the domain, accomplishing a sort of requirements analysis similar to what is accomplished during the Concept Exploration phase.

- o *Architect the Domain* develops and validates a feasible architecture, as would be expected at the conclusion of the Demonstration and Validation phase.

- o *Develop Component Assets* is simply the Full Scale Development of the component library.

This analogy is useful in addressing the relationship of the domain analysis process to the overall software development process. Two options are identified:

- o As a precursor activity to future system developments in the domain

- o As a parallel activity to a development of an instance of a system in the domain

As a precursor activity to future system developments in the domain, the entire domain analysis process is accomplished first and then provides its reusable products to future developments of system instances. This is depicted in Figure 3.2-1. Though not shown on this diagram, the domain model, architecture and library will be refined as each new system is developed and as the underlying domain changes and evolves.

The advantage of this approach is that the strategic, long-term view of reusability can be analyzed and assessed for the application domain. With a generic architecture and defined interfaces, reusability can be reinforced with policies and standards. Future system procurements can specifically address adherence to the generic architecture and use of the existing component base. The risk with a precursor activity is that investments will not have the expected return because of changes in the domain or underlying technologies or because of bad decisions or inadequate analysis.

As a parallel activity to a system development in the domain, the three domain analysis phases parallel the first three system phases, as shown in Figure 3.2-2. During each of the system development phases, both the generic and the specific models, architectures and software are considered. Since the domain analysis activity employs many of the same system architecture and design techniques, it is believed that synergies could be discovered that will lower the incremental cost for such a parallel activity. These synergies, of course, would be greater if both activities were accomplished by the same organization.

A parallel activity has a short-term economic advantage because of the engineering leverage and because of the more immediate benefits recognized in the first system instance. Development of a complete, deployed system will provide the best demonstration of the potential of reusability in the application domain. The risk of a parallel activity is that the domain analysis will be biased by the first system instance and may not have the same level of benefit for future system developments. This will require additional effort to maintain and evolve the domain analysis products as the requirements of these future systems are adequately considered and analyzed.

There are, of course, may other variants that could be derived to address particular constraints or acquisition realities. Domain analysis should become a strategic aspect of program planning.

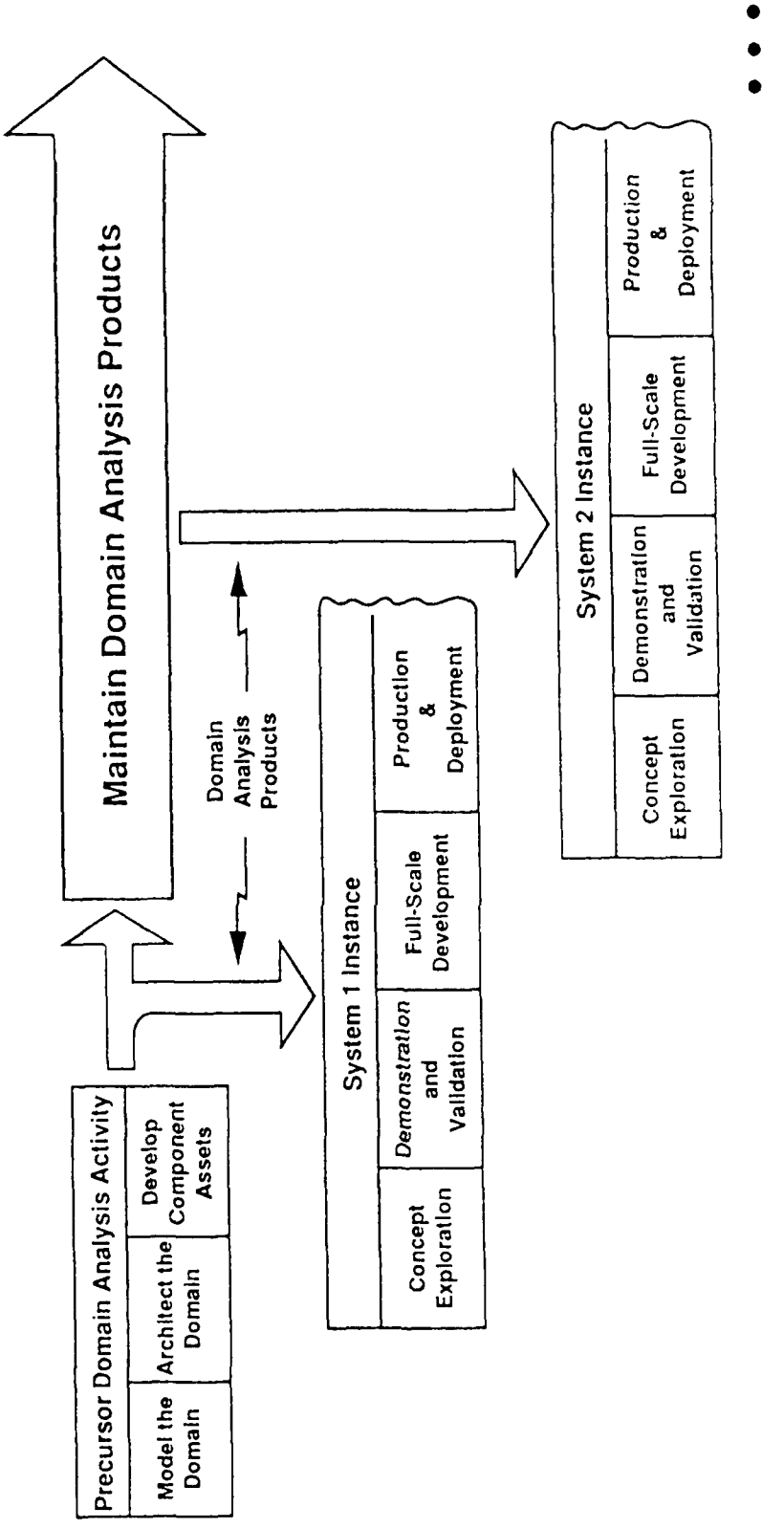


Figure 3.2-1 Relationship of Precursor Reuse Activity to System Development Life Cycle

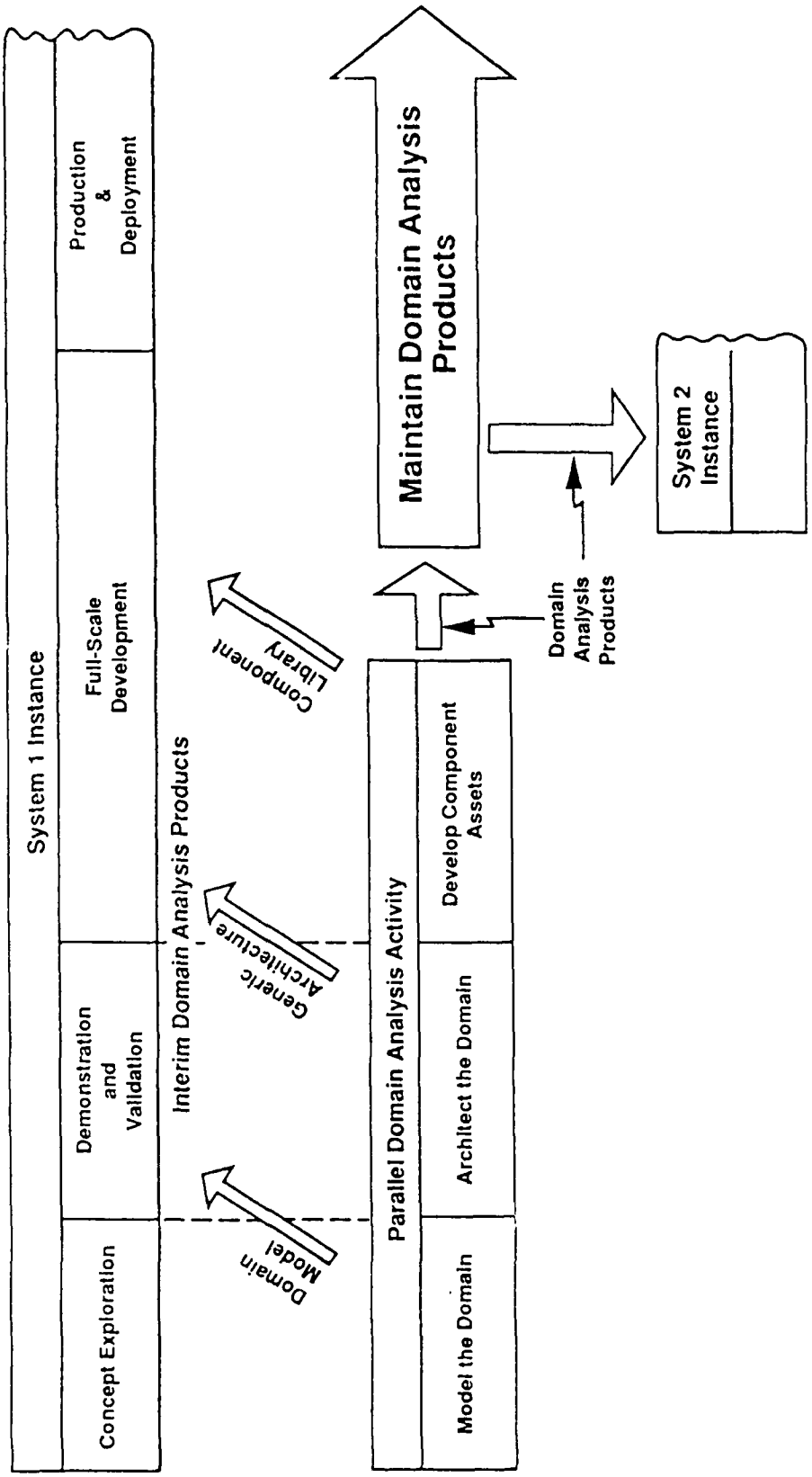


Figure 3.2-2. Relationship of Parallel Reuse Activity to System Development Life Cycle

3.3 Process Model for Domain Analysis

The domain analysis process is directed toward bounding and analyzing an application domain to produce a set of reusable domain analysis products, as depicted in Figure 3.3-1. Inputs to this process include knowledge of previous systems in the application domain and planning documents that provide insight into future needs and evolutions of the application domain. The domain analysis process must be planned and scoped within the context of programmatic goals and constraints.

The recommended domain analysis process consists of three phases, shown in Figure 3.3-2. This figure also shows the various domain analysis products and the relationship between the phases using a notation borrowed from SADT [MAR88]. The *Model the Domain* process produces the following products:

- o *Domain planning document* that bounds the domain, scopes and plans the domain analysis activities, establishes guidelines and standards, and assesses the costs, risks and benefits of the effort.

- o *Domain dictionary*, precisely defining a controlled vocabulary for the domain analysis activities.

- o *Validated domain model* representing the specification of application objects and their interaction. This model encapsulates the knowledge of commonalities and differences amongst the family of systems considered in the domain. The domain model also includes a specification of additional complementary application requirements and constraints and guidelines for using the model.

- o *Domain validation products* consisting of scenarios, simulations and executable specifications that are used to validate the correctness and completeness of the domain model.

These products are input to the phase to *Architect the Domain*. The goal of this phase is to transition the abstract domain model into a concrete generic architecture or family of architectures that form the blueprint for the domain family of systems. The outputs of this phase are:

- o *Validated generic architecture(s)* and associated tradeoffs and systems analyses for the application domain family of systems. The generic architecture product includes guidance for applying it.

- o *Generic architecture validation products*, specifically simulations, prototypes and partial implementations that demonstrate the architecture.

The generic architecture defines the software components to be developed in the phase *Develop Component Assets*. This final phase produces the following products:

- o *Component classifications* that identify and categorize that software components that will be developed.

- o *Interface and protocol specifications* for key interfaces to components and component sets and guidelines for using them.

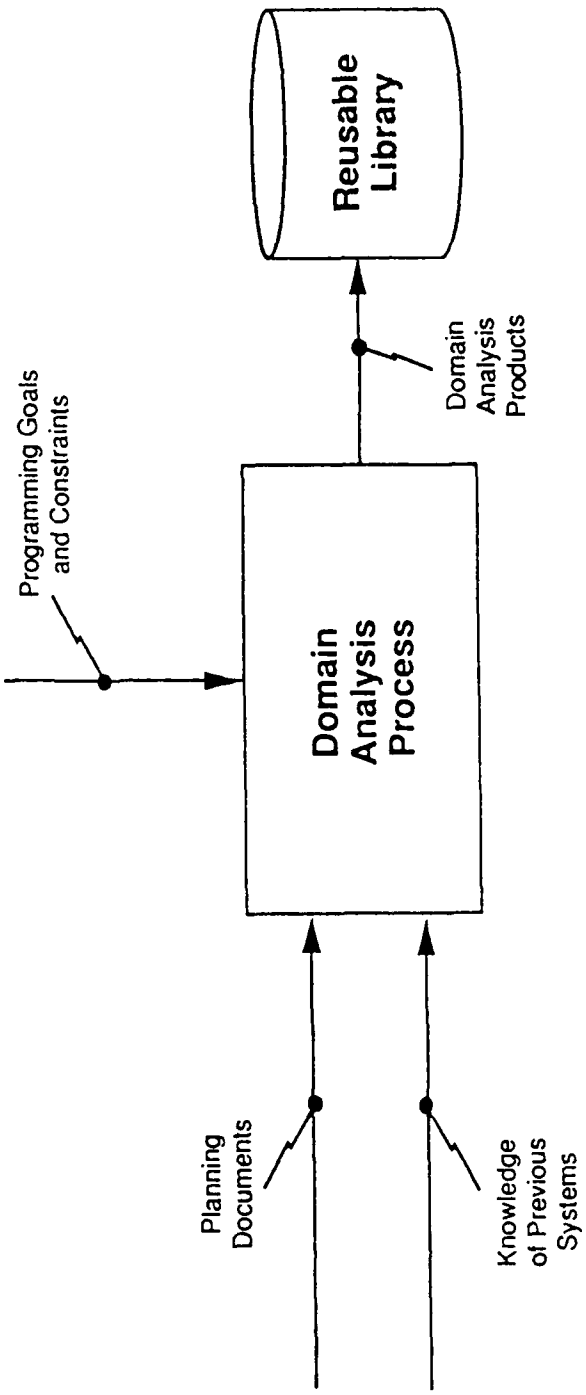


Figure 3.3-1. Domain Analysis Process.

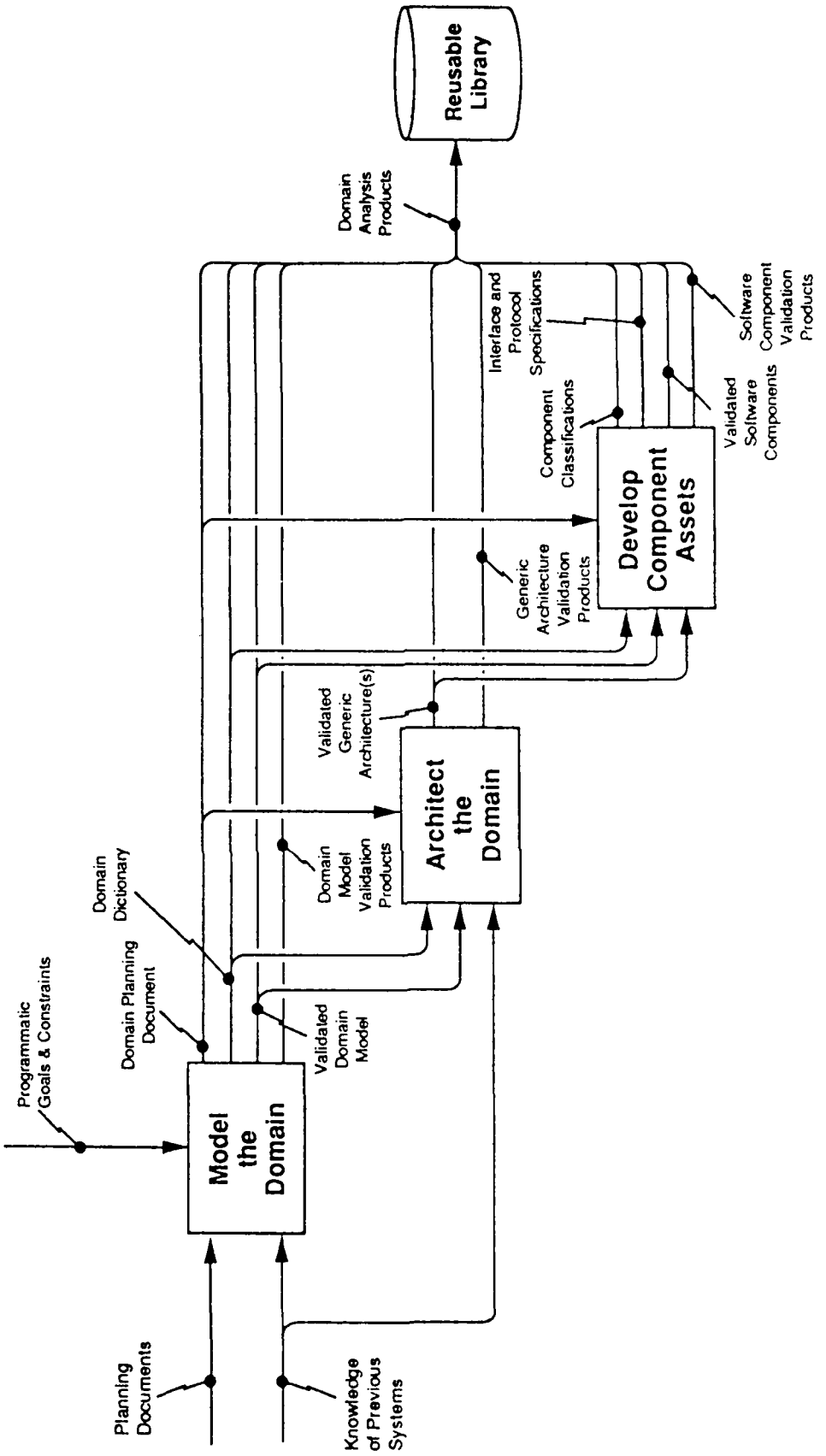


Figure 3.3-2. Domain Analysis Phases.

- o *Validated software components* for adaptation and reuse in developing systems in the application domain. A software component includes the code, its adaptation mechanism, documentation, classification characterization, tradeoffs and rationale, and guidance for reusing it.

- o *Software component validation products*, representing the test suites used for component validation. This may include test plans and procedures, test cases, test data, test results, proofs or demonstrations.

Note that *all* of the domain analysis products are potentially reusable during the development of system instances. The domain dictionary and domain model form the basis for requirements analysis for a specific system. The generic architecture similarly is the starting point for system design. Interface specifications, prototypes and simulations are also potentially reusable. Thus, the domain analysis results in *reusable systems engineering*, in addition to reusable software components.

Development of system instances starts with the domain models and architectures as a baseline rather than starting from scratch. The development process then focuses on elaborating refinements and deviations from this baseline. In this way, the domain analysis activity captures the knowledge and skills of domain experts and expert systems engineers for future reuse.

Eleven steps are defined for the three phases of the domain analysis process. Figure 3.3-3 shows the four steps that comprise the *Model the Domain* phase:

1. *Scope the domain analysis*, generating the domain planning document that will guide the domain analysis activities.

2. *Perform domain commonality analysis*, resulting in an object-oriented domain model.

3. *Perform domain adaptation analysis*, that annotates the application objects with adaptation requirements to accommodate the differences between systems in the application domain.

4. *Validate the domain model* through analyses, scenarios and simulations.

Three additional steps comprise the phase *Architect the Domain* (see Figure 3.3-4):

5. *Identify enabling component base*, specifically those general or multi-domain components that were not discovered in the domain modeling activity.

6. *Develop generic architecture(s)* that may be used for the family of systems in the application domain.

7. *Validate the generic architecture(s)* through prototypes and partial implementations.

The final four steps comprise the *Develop Software Component Assets* phase (see Figure 3.3-5):

8. *Develop software component classification(s)* that identify and categorize the software component assets to be developed.

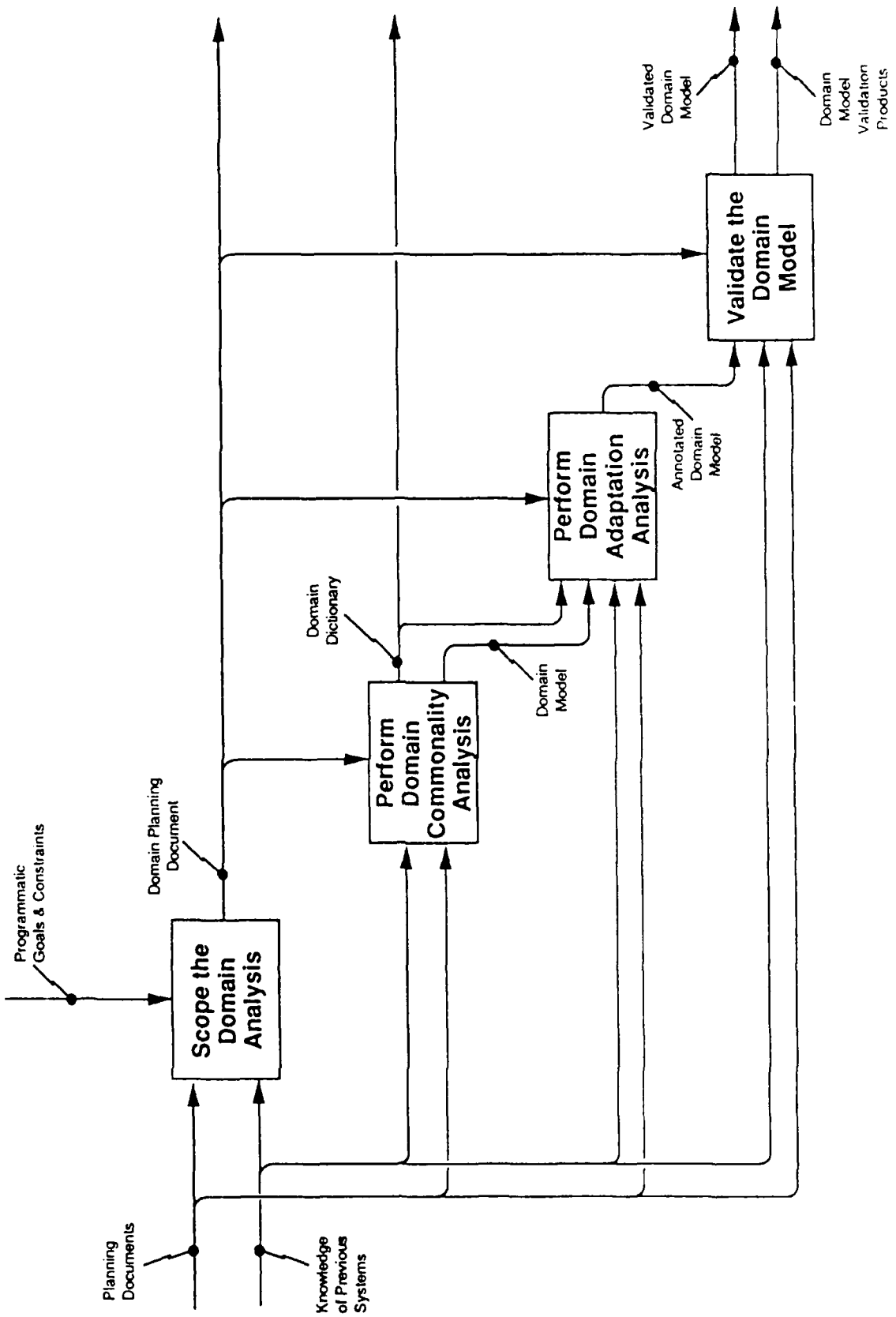


Figure 3.3-3. Model the Domain

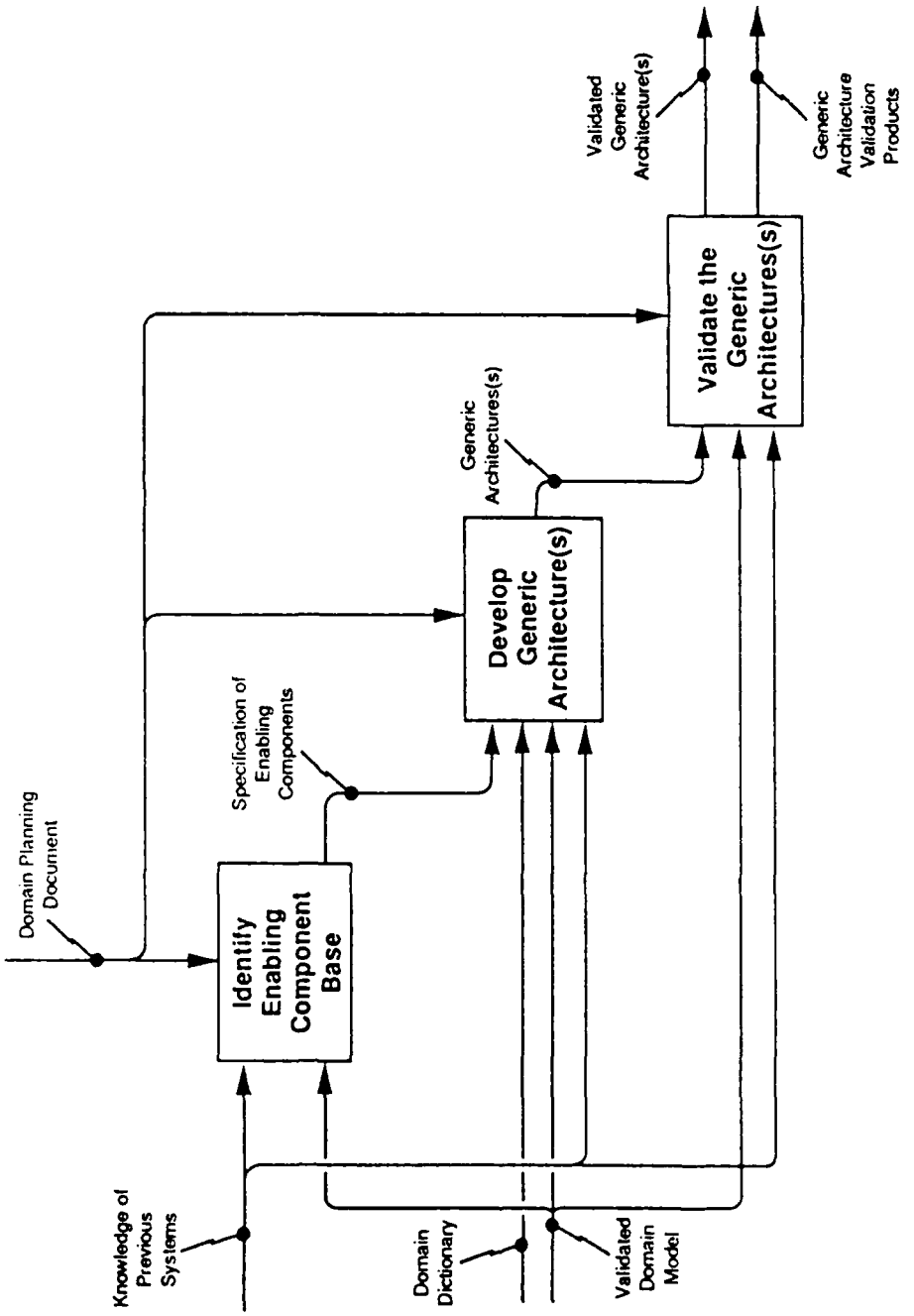


Figure 3.3-4. Achitect the Domain.

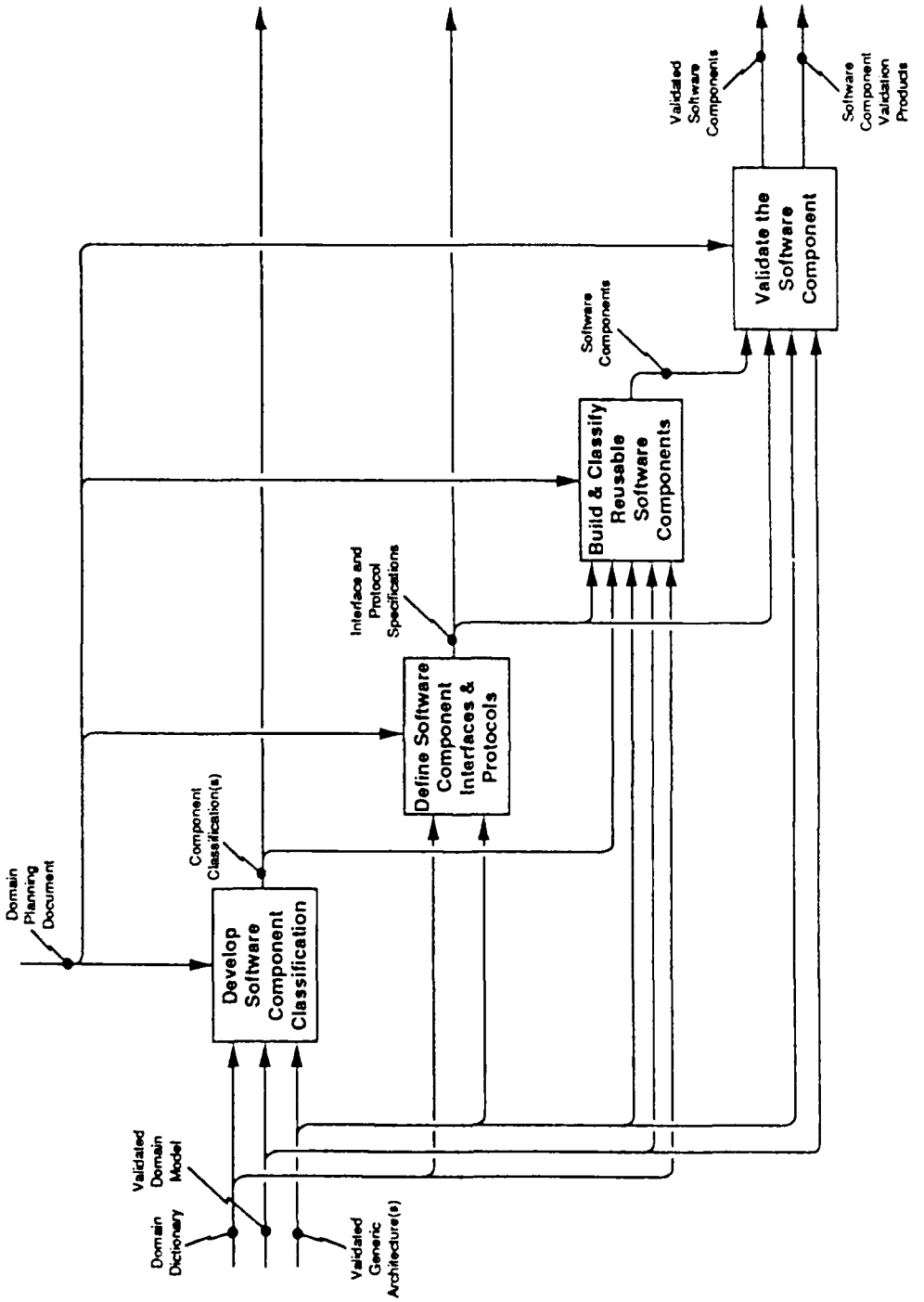


Figure 3.3-5. Develop Software Component Assets

9. *Define software component interfaces and protocols* for key interfaces in the generic architecture. These interfaces will precisely define the components and component sets to be developed.

10. *Build and catalog reusable software components* for reuse in building system instances in the application domain.

11. *Validate the software components* using standard testing and verification techniques.

The following subsections describe each of the eleven steps.

3.3.1 Scope the Domain Analysis

This first step seeks to scope the domain analysis and develop a detailed plan. The output is a Domain Planning Document that includes:

- o Domain history and rationale
- o Sources of knowledge and information about the domain
- o Scope of the application domain to be considered
- o Scope of the domain analysis activity
- o Plan for the domain analysis activity
- o Guidelines for the domain analysis activity
- o *Cost-benefit and risk analysis*

The Domain Planning Document begins by briefly establishing the context for the document by defining the domain and providing domain history. The rationale for the domain analysis and expectations are stated. Specific issues are identified to be addressed. Directly or indirectly involved agencies and organizations are identified.

Information sources are identified for the application domain. It is important that sources be identified for information regarding both previously developed existing systems in the application domain and for future, contemplated systems. Analysis of previous systems is valuable for acquiring a basic understanding of the application domain. However, the primary focus must be to understand the needs, requirements, commonalities and differences of *future* systems. Potential information sources include:

- o Application domain planning documents
- o Programmatic/acquisition documents
- o Application domain needs and technology studies
- o Concept exploration studies for future systems
- o Documentation and prototypes demonstrating future system concepts
- o Existing systems (and their documentation) in the application domain

o Identification of specific domain experts

o Identification of existing reusable assets to be considered

The scope of an application domain is defined by specifying what is considered as part of the domain and what is considered outside the domain. This scope includes:

o System boundaries in the domain

o Required or typical system interfaces

o Constraints on implementations

o Projections regarding future systems

o Knowledge about past systems

o Subjective assessments regarding the commonalities and differences between system instances in the application domain

o Identification of known subdomains or of overlaps with other application domains

o Various assumptions regarding risks or unknowns

The techniques applied are exactly the same as those used to scope a particular system instance, only now a *family of system instances* are considered. The necessary insight can be obtained from domain experts and from future planning documents, documentation about previous systems in the domain, and from knowledge of overall programmatic goals and constraints. Attention must be given early to clearly identifying the boundaries of the application domain, addressing those "fuzzy" areas where there is no clear agreement on whether a particular capability is in the domain or not.

The best way to precisely scope the domain is to specifically enumerate the system instances that comprise the application domain family to be addressed. These system instances will define the scope of variability that the domain analysis products will be expected to address. This will allow a more accurate cost-benefit assessment to drive the domain analysis decision process.

Within a domain, specific subdomains may be identified if the whole domain is large or complex, to focus on areas of highest payoff first, to address loosely coupled subsystems, or to allow parallel domain analyses by separate teams. Identification of subdomains may be important in making a domain analysis tractable for large applications.

In addition, the scope of the domain analysis *activity* must be addressed. Realistically, cost, schedule or other programmatic constraints may not afford the "ultimate" domain analysis. Lack of information or lack of understanding may make the ideal analysis intractable. Potential ways to further scope the domain analysis include limiting the number of systems to be considered or by considering subdomains.

Limiting the number of systems instances to be considered in the family of systems is a decision to take a shorter term view of the domain. This may affect the longevity of the domain analysis and thus ultimately affect the economic payback for the effort. Such

scoping may be appropriate for domains that are changing relatively fast or whose evolution is not well understood.

Limiting the domain analysis to subdomains focuses the activity to areas of highest potential reuse payoff. These would be identified based upon an analysis to identify those areas with maximum commonality and where the differences between systems can be effectively handled with known adaptation techniques. Such an approach could be applied to allow domain analysis to proceed in a piecemeal fashion, incrementally addressing different aspects of a large application domain over time.

The costs and the benefits of any type of scoping must be well understood. The effect of domain scoping decisions on each of the future system instances must be assessed in light of future funding and schedule constraints. There is also a danger that domain scoping will affect efforts in other related domains, possibly limiting cross-domain reusability between them.

A detailed plan for the specific domain analysis activity is produced that documents the scope of the effort and sets the appropriate milestones, deliverables and budgets. Resources that must be addressed include personnel, domain information, tools, computing resources (for development, prototyping, simulation, documentation, etc.), and facilities. Both the availability and the quality of resources must be addressed (for example, whether a domain expert is available and how much experience he or she has).

Guidelines for the domain analysis activity define how to perform the domain analysis. The specific process to be followed and the techniques, languages and representations to be used are identified. Tools to support the process are selected. Specific analyses to be employed are described and issues to be addressed are stated.

The Domain Planning Document includes a cost-benefit analysis based upon the detailed cost estimates for the domain analysis and the project benefits of reusability to be reaped in the development of the various system instances. Cost benefit analysis should be used to tradeoff various domain analysis scoping approaches. A decision to proceed (or not to proceed) with the domain analysis should be made based upon justifiable economic reasons. As additional information is discovered as the domain analysis proceeds this analysis should be updated and the decision reassessed. A risk analysis is accomplished that identifies and assesses potential risks that could hinder reusability and details plans for abatement of those risks.

Developing the Domain Planning Document is an important and sizable effort, involving a number of scoping, analysis, tradeoff and planning activities. It is critical that enough attention be paid to this first step.

3.3.2 Perform Domain Commonality Analysis

Domain commonality analysis is accomplished through the development of an object-oriented domain model. The commonality analysis activity produces two primary products derived through detailed analysis of the domain:

- o Domain dictionary
- o Domain model

It is important that the analysis consider both future system requirements, as may be found in planning documents and studies, as well as existing systems in the domain. The

domain analysis is meant to address the needs of *future* systems. Consideration of existing implementations must be tempered by analyzing future needs.

A good place to start this activity is to develop a domain dictionary of terms as the domain is researched. This dictionary will define the terminology or language of the application domain and serve to ensure consistency throughout the domain analysis steps. The domain dictionary should be baselined and carefully configuration controlled. The domain dictionary consists of:

- o Terms and definitions
- o Relationship of terms

The identification and definition of terms is a conceptually simple activity that often takes considerable effort to coalesce a set of accepted definitions. The relationship of terms may take many forms. The most common is a thesaurus that defines synonyms, more specific and more general terms, and related terms. A more formal, but more complex, way to related terms is by using a semantic network. If a generative approach to reuse is being considered, as with Draco [NEI81], a domain language could be defined. While a simple, but precise, dictionary format is considered sufficient, more ambitious domain engineers may employ more complex representations.

The primary focus of the commonality analysis is to develop a domain model. An domain model consists of:

- o Object-oriented model
- o Other complementary domain requirements and constraints
- o Traceability to the domain definition documents
- o Tradeoffs and rationale for the domain model
- o Guidelines for using the domain model

Object-oriented knowledge modeling techniques are recommended as the foundation of the domain model. Such techniques specify the various application domain objects in terms of the following:

- o Objects
- o Operations
- o Stimuli/responses
- o Transactions
- o Attributes
- o State
- o Relations
- o Constraints

The following discussion provides an overview of these concepts. It is meant to provide an introduction to the concepts rather than present a particular method. The material should prove helpful to the reader evaluating candidate object modeling techniques.

An *object* combines the concepts of function and data, by encapsulating a set of data and providing a set of predefined operations to manipulate and access the data. The concept of an object for our context is exactly the same as the notion of an object in everyday life. An object is a "thing," such as:

- o Tangible things (e.g., vehicle, airplane, computer)
- o Roles played by persons or organizations (e.g., owner, client, broker)
- o Incidents, an occurrence or event that happens in time (e.g., flight, accident)
- o Interactions, forming a tangible contract or transaction (e.g., purchase)
- o Specifications, standards or definitions (e.g., model specification, inventory description)

A system is itself an object. It is composed of lower level objects. These objects represent the things or events in reality which are of interest to the system. They are things close to reality about which you want information. Since the system objects are objects in reality, it is of utmost importance to reflect them in the developed system.[JAC87]

In many cases, when we refer to an object of interest to a system, such as a user, a plane, or a site, we are actually referring to a collection of similar objects, specifically, many users, many planes or many sites. These *object classes* serve to specify the common characteristics, or attributes, of each object, and to describe common behavior, or operations. Thus an object class denotes a set of similar, but unique, object instances.

An *object instance* is a separable and distinguishable member of an object class. An object instance has behavior defined by the abstract operations of the object class and has specific attribute values that distinguish it from other instances and define its state (see Figure 3.3.2-1). An object has the following characteristics:

- o Is characterized by the operations it provides
- o Responds to stimuli from other objects that request operations
- o Participates in system transactions that define series of interactions with common application purposes
- o Encapsulates attributes that represent the data pertinent to that object
- o Has state, as defined by the values of its attributes
- o May be encapsulated, or nested within other objects
- o May be related to other objects
- o May have constrained states or behavior

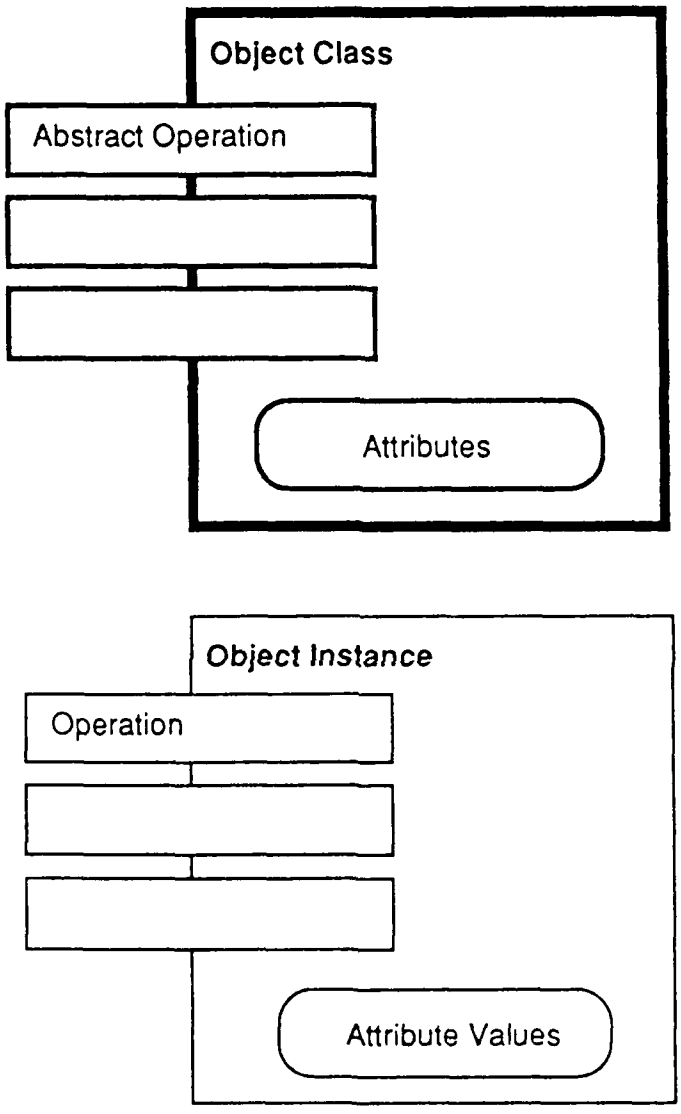


Figure 3.3.2-1. Static Model of an Object

An object's *operations* defines the object's capacity for action, response or functioning. An object's operations completely characterize its external behavior. Some of the operations may return data values that give information about the object. Others may change the encapsulated attribute values. [BAR86]

A *stimulus* is an external request for an operation made upon an object. Operations are invoked upon receipt of an external stimuli (see Figure 3.3.2-2). A stimulus may generated as a result of an event in the object's environment or be an operation request from another object. A stimulus may include necessary input data to the object. For each stimulus, the object produces a *response* (see Figure 3.3.2-3). This response may include:

- o Null (no action)
- o Return of data to the requesting object
- o Update of the object's attribute values, thereby changing its state
- o Interaction with other objects

Transactions are behaviorally related sequences of stimuli and responses. In many cases, in order to carry on a complex action, many stimuli and responses are needed, potentially involving many objects (see Figure 3.3.2-4). These sets of related operations are termed transactions. A transaction represents a series of interactions with a special purpose. The collected description of a system's transactions constitutes its total behavior. [JAC87]

Attributes define the data pertinent to each instance of an object. Attributes, in a general sense, are the variable properties that characterize or differentiate one member of that class from another. [ABB86] Typical attributes might include measures of an object (e.g., size, weight, power), labels for the object (e.g., name, serial number), or characterizations and other descriptors of objects of that class (e.g., date installed, access rights).

More specifically, attributes represent the "needed remembrance" of an object. [COA89] An object needs to remember those things that allow it to behave properly and successfully interact with other objects. There are several types of attributes:

- o Naming attributes - that uniquely distinguish or identifies each instance of an object class
- o Descriptive attributes - representing the intrinsic characteristics of the object
- o State data attributes - that encapsulates stimulus histories to define the internal state of the object
- o Referential attributes - that relate an instance of an object to other object instances

The specific response of an object's operations may depend on the values of its attributes. These attribute values denote the *state* the object is in. Hence we may consider objects to be equivalent to finite state machines, which given a state (i.e., initial attribute values) and an input stimulus, will produce a response and a new state (i.e., updated attribute values).

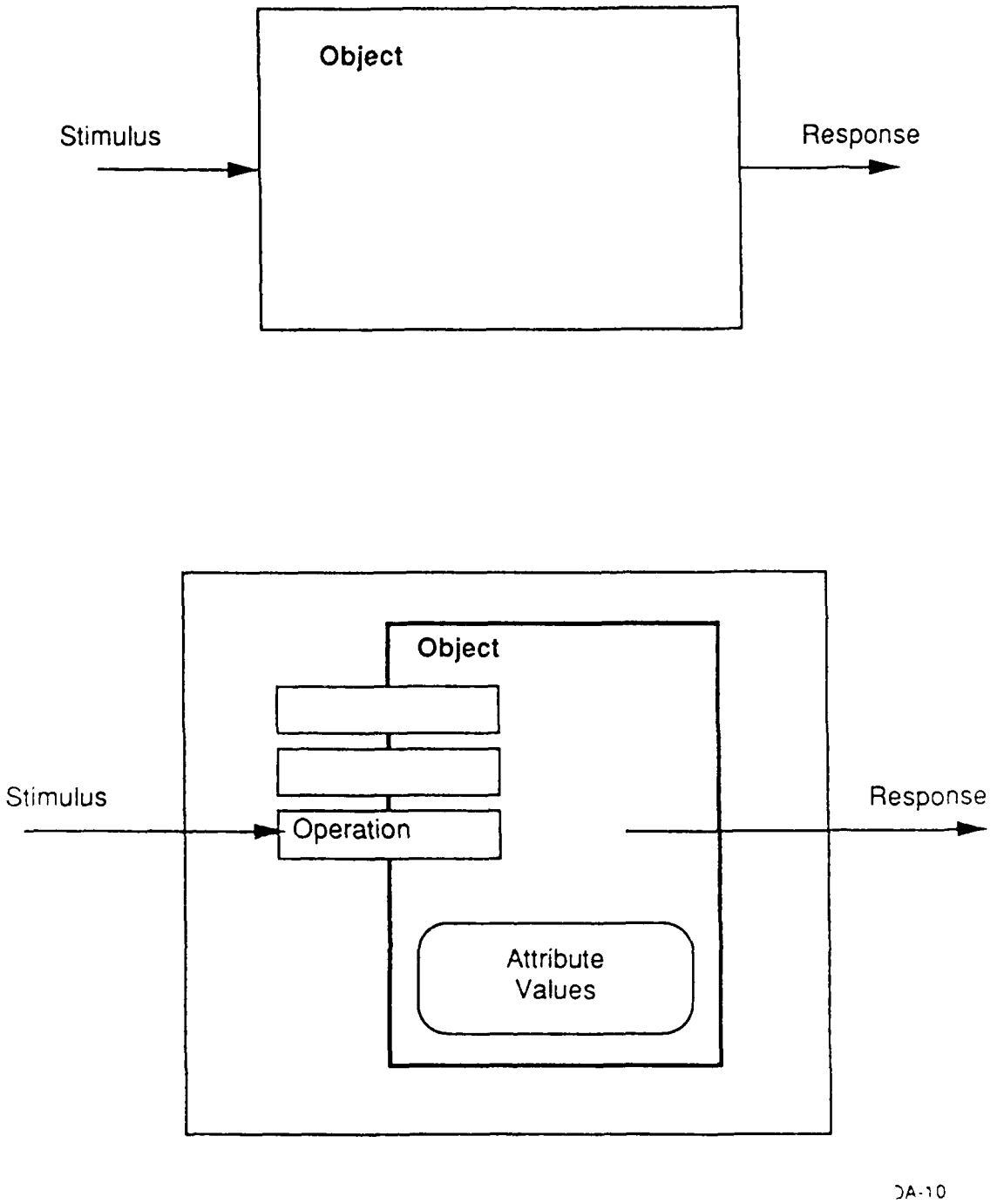


Figure 3.3.2-2. Dynamic Model of an Object

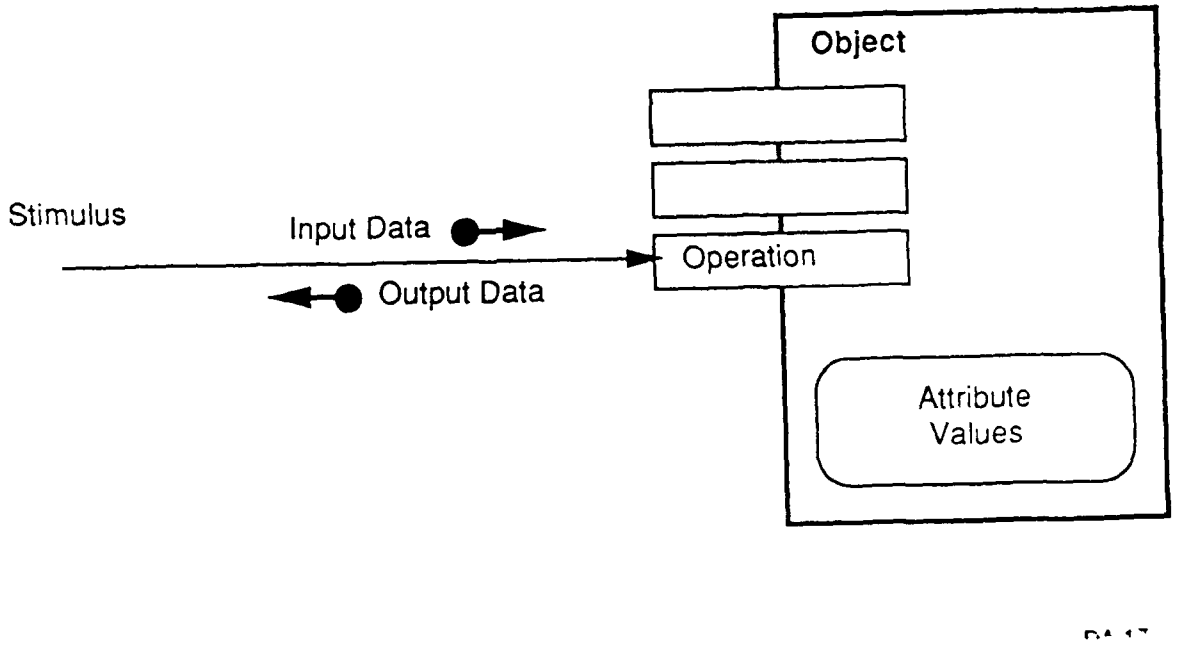


Figure 3.3.2-3. Stimulus Interaction Model

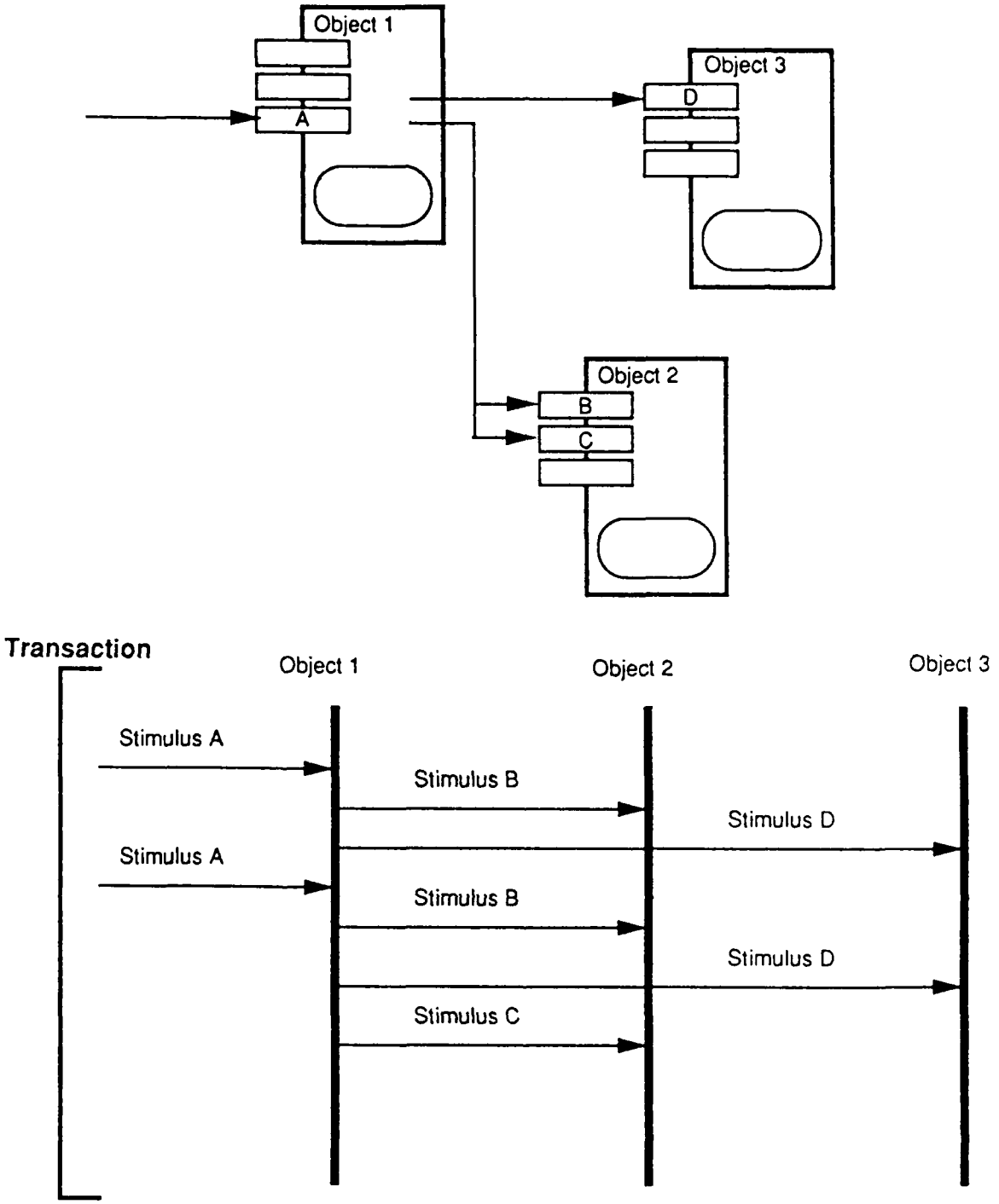


Figure 3.3.2-4. Object Interaction

Objects may be nested to reflect the fact that often in real life systems a complex object is composed of many subobjects. The subobjects contribute to the state and behavior of a parent object. The operations on these subobjects are utilized to provide the services of the parent object (see Figure 3.3.2-5). However, another object requiring the services by the parent object does not know, nor need to know, about the existence of subobjects. [JAL89] It is important to note that an object's attributes may be represented as nested subobjects.

A *relation* represents a logical, named, association between objects having a well-defined role in the application domain. Some relations may be static (e.g., a physical assembly) or may be dynamic, only occurring at a specific period in time (e.g., as in passengers in a plane). Figure 3.3.2-6 illustrates one symbology for object relations. Objects may be related or associated in a number of ways, including: [COA89]

- o Assembly - "part-of" relation denoting a whole and its component parts
- o Classification - "is-a" relation denoting generalization/specialization relationships between object classes
- o Occurrence - a general relation between object instances
- o Interaction - a relation between an object which sends a stimuli to an object that receives the stimuli

Each of elements in the domain model may be annotated with additional *constraints*, representing conditions that must be met at particular points in time or over specific periods of object activity. Constraints are used to denote the limitations of the domain on its objects and their behavior. They serve to rule out anomalous states. [LON86] Constraints represent a general capability to declaratively assert conditions that must be true for a particular element or set of *related elements*. As such there are a number of possible forms constraints may take, including:

- o Existence of attribute values or relations
- o Cardinalities of attributes or relations
- o Constraints on allowable values of an attribute
- o Condition relating multiple attribute values that constrains the values that these attributes may hold
- o Preconditions to an operation
- o Postconditions to successful completion of an operation
- o Temporal relationships of objects and operations

The domain modeling technique should be to able to analyze the application from different perspectives. This is a powerful way to reason about and understand systems. There are two kinds of perspectives that are of particular use in understanding and analyzing objects:

- o Static and dynamic perspectives of objects
- o External and internal object perspectives, and inter-subobject perspective

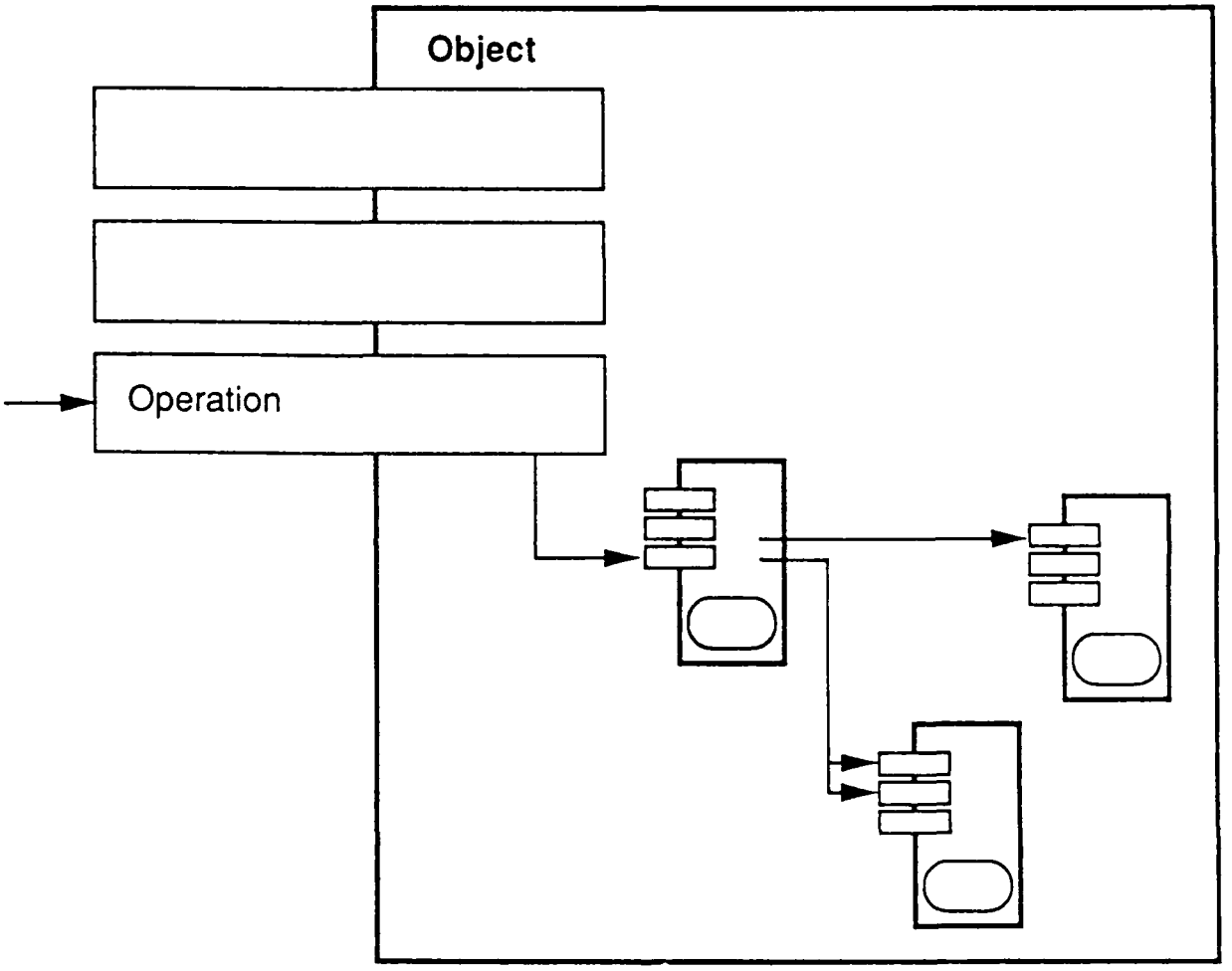


Figure 3.3.2-5. Nested Objects.

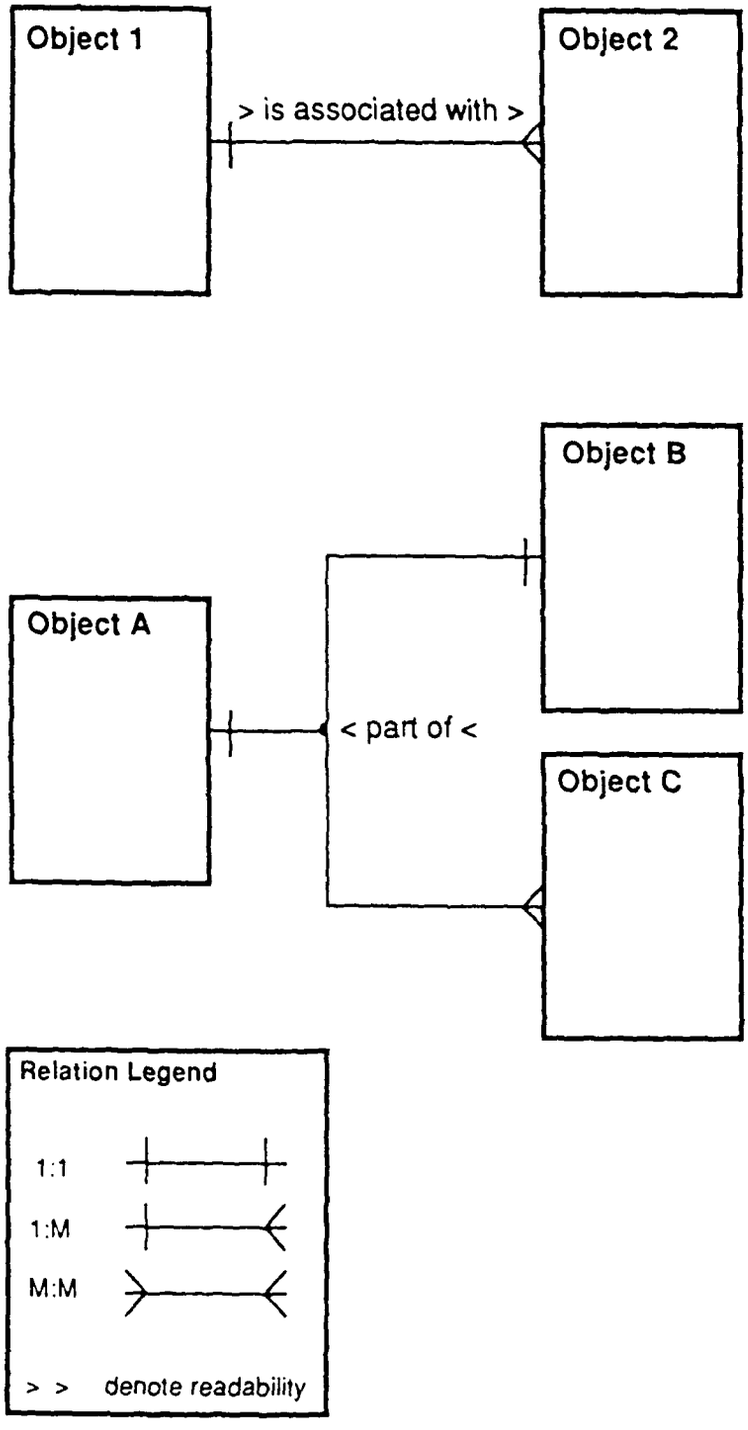


Figure 3.3.2-6. Relations between Objects.

The static and dynamic perspectives look at objects within the context of time. A static perspective analyzes the object at rest, discovering facts that are, for the most part, invariant over time. Static analysis centers on characteristics and associations. The dynamic perspective analyzes the object in action, over time. Dynamic analysis focuses on behavior and interaction. The dynamic perspective addresses real-time performance.

Another kind of perspective looks at objects with different degrees of internal visibility. Clearly, every object can be viewed in two different ways: an outside perspective and an inside perspective. These perspectives are based upon the very principle of data abstraction. Whereas the outside perspective serves to capture the abstract behavior of the object, the inside perspective indicates how the behavior is implemented. [BOO87] If an object is composed of subobjects, a third degree of internal visibility is to analyze the relations and interaction between these subobjects.

The static and dynamic perspectives, when applied to the external, internal and inter-object perspectives, provides an organization for modeling objects. This organization, shown in Table 3.3.2-1, provides the foundation for object-oriented analysis.

The external object model considers only those aspects that can be viewed from the outside. Statically, this entails identifying the object name, its stimuli and the application transactions they support, and the object's operations. Dynamically, the behavior and interactions of the object are analyzed only in terms of those aspects that are viewed externally. The dynamic response of an object is described in terms of its stimulus history.

The internal object model statically defines the attributes of the object that the object must remember. These attributes serve to encapsulate the stimulus histories in a form that captures the state of the object. As a result, a dynamic perspective of the internal object model bases its characterization of behavior of the object on the attributes or state of the object. This kind of behavioral analysis results in an operational specification.

The inter-subobject model statically defines the subobjects that are nested within the object and defines the static relations and interaction paths. From a dynamic perspective, the interactions of these subobjects are analyzed in terms of their participation in the system transactions.

A combined graphic and textual object modeling technique is recommended for the object modeling. The object-modeling process will, by its very nature, identify and elaborate the objects that are common in the application domain. The objective is to develop a type of generic requirements model that represents the applications. By modeling both the system objects both statically and dynamically, commonality is addressed both structurally and behaviorally.

To support future architectural analyses, it is important to document additional complementary requirements and constraints for the domain, such as:

- o Physical requirements
- o Locality requirements
- o Performance requirements
- o Fault tolerance requirements
- o Survivability requirements

Table 3.3.2-1 Object Perspectives

	Static Perspective	Dynamic Perspective
External Object Model	• Stimuli • Responses • Transactions • Operations	• External object behavior • External interaction models
Internal Object Model	• Attributes	• Operational specification of behavior
Inter-Subobject Model	• Subobjects • Subobject relations • Interaction paths	• Subobject interaction models

- o Security requirements
- o Quality requirements
- o Personnel requirements
- o Logistics support requirements
- o Special cost requirements

The domain model plus these additional requirements represent the requirements for the family of systems in the application domain.

Traceability to specific domain definition documents and sources must be carefully maintained. The documentation includes tradeoffs and rationale for the domain model and the complementary requirements. Guidelines for using the domain model are included as an introductory section.

3.3.3 Perform Domain Adaptation Analysis

While the previous step models the *commonality* between system instances in the domain, this step seeks to determine the *differences* between system instances. The result of this step is an annotated domain model that specifies for each domain object:

- o The adaptation requirements imposed by domain
- o The adaptation mechanism recommended to meet the requirements

Adaptation analysis is critical to deriving a generic architecture and component library that can adapt to future system needs. This analysis of required adaptation may be based upon mission, threat, domain, or system planning information. Additionally, trends observed from an analysis of changes or evolutions of previous systems may be helpful.

The domain adaptation requirements are stated by simply annotating the various objects in the domain model as to their anticipated adaptation requirements, including:

- o Flexibility in operation
- o Mission adaptation (needs, threats, etc.)
- o Environment/site adaptation
- o Platform adaptation
- o User adaptation
- o Technology adaptation (e.g. to accommodate new computing or sensor technologies)

This is accomplished either by annotating the object descriptions directly, or by defining the overall domain adaptation requirements and allocating them to specific objects (e.g., using a traceability matrix).

Next, the potential adaptation mechanisms are identified that can accomplish the required level of adaptation per object. This analysis looks toward possible implementation strategies that could be employed *on an object-by-object basis* to accommodate the desired differences over instances. For example, this could include:

- o *Fixed* - no adaptation needed
- o *Parametrized* - adapted using call parameters
- o *Generic* - adapted through generic object, type and/or subprogram parameters
- o *Data-driven* - typically from file or database rather than parameters
- o *Language-driven* - adapted using a language syntax
- o *Template algorithms* - used by "completing the blanks" in the component
- o *Variant selection* - adapted by selecting from a few components or component sets with standard interfaces but different implementations
- o *Specialized* - adapted by adding characteristics or behavior to a generalized class definition
- o *Generator/constructor-based* - requiring a tool for generation of adapted components
- o *Custom* - requiring a complete new implementation for each system instance

Upon completion of this step, both the commonalities and the differences of projected future system instances are well understood for the family of systems in the application domain.

The domain model should be sufficiently rigorous to form the basis for a simulation of the system. While such a simulation would entail considerable effort for a large system, it could be constructively used during the domain modeling activity to refine the object model and to explore adaptation mechanisms. Simulation provides an excellent means for analyzing the effects of future system environment changes (i.e., mission, threat, theater, etc.) and for performing "what-if" analyses.

With additional rigor and automation, it is possible that the domain model be constructed using a representation that supports execution of the model. Such an executable model represents a prototype of the system that provides the highest level of model validation.

The validated domain model provides a tangible baseline to base a more detailed cost-benefit analysis. The commonality and adaptation analyses can be used to predict the level of reusability potential in the application domain. The necessary adaptation mechanisms for each domain model object provides the basis for estimating the costs of achieving that level of reusability. Areas of risk can similarly be more accurately assessed.

These analyses provide the rationale for a go/no-go decision or for additional domain analysis scoping to maximize the cost-benefits that will be achieved. Scoping could

accomplished by future efforts addressing only a subset of common objects (i.e., subdomains) or by limiting the types of adaptation mechanisms provided.

3.3.4 Validate the Domain Model

The domain model is validated by independent review that considers the requirements for specific future system instances. The output of this step is a validated domain model plus the various domain model validation products that are a byproduct of the activity. These validation products could include:

- o Operational scenarios
- o Simulations
- o Executable model prototypes

The validation may be accomplished by "conceptually constructing" future instances from the objects identified in the domain model. Operational scenarios trace the interactions of objects for system transactions for a future system instance. For each operational scenario, reviewers analyze the object model and the adaptation analysis to verify that these specific future systems can be accommodated.

3.3.5 Identify Enabling Component Base

This step seeks to identify additional underlying components that enable an efficient generic architecture to be developed. The output of the step is a specification of these enabling components and technologies.

The domain model that is developed in the previous steps views the system from an operational perspective. It is expected to discover all of the "domain-dependent" objects (i.e., the "vertical" domain [MCN86a]). It is natural that this perspective may not discover lower level reuse potentials that *enable* the application to be implemented easier. These undiscovered objects will typically represent "application-independent" or "horizontal domain" [MCN86a] objects. For example, it would be unusual for a domain model to identify an operating system object. Yet such an object is likely necessary to enable the implementation. The potential for these *enabling components* is analyzed prior to developing a generic architecture.

Common examples of potential enabling components are in the areas of:

- o Mathematical operations
- o Data structures and manipulations
- o Information management subsystems or capabilities
- o User interfaces
- o Communications

These enabling components are discovered in a number of ways:

- o Browsing the library of existing components

- o Analyzing the domain model, identifying enabling components that could support each type of object
- o Further elaborating the domain model, or portions of the domain model, to greater levels of detail
- o Analyzing the design of previous systems
- o Investigating components identified in related horizontal domains
- o Based upon the experience of the domain analyst in implementing similar systems

Compatible with the previous analyses, the enabling components identified should assume an object-oriented implementation. Adaptation analysis is performed for each enabling component object to identify the required mechanisms for adaptation. The combination of the objects identified in the domain model plus those objects identified as enabling components form the basis for synthesizing a generic architecture in the next step.

As part of this activity, the various underlying hardware and software technologies must be understood for the projected timeframes of implementation for the system instances. For a family of systems that spans a significant period of time this may require performing a technology forecast. Hardware and software technology forecasts can be obtained by searching the literature for such projections by experts in the various fields. Technology forecasts may also be obtained from marketing research firms or from major vendors in the market.

A software technology forecast would entail investigating future:

- o Languages
- o Automated environments
- o Software architectures
- o Software productivity and reliability trends
- o Communications and interface standards

A hardware technology forecast would include projections of future:

- o Underlying semiconductor capabilities (speed, size, weight and power)
- o Computing architectures and characteristics
- o Communication technologies
- o Communications and interface standards
- o Sensor and control technologies

Software technology has, in the past, evolved relatively slowly, with significant differences only being seen on a decade by decade basis. Hardware technology evolution has been more rapid, with significant advances occurring every three to five years. These

frequencies would indicate that *every* domain analysis needs to do some form of technology forecast.

3.3.6 Develop Generic Architecture(s)

A generic architecture, or family of generic architectures, are derived for the application domain. The resulting generic architecture(s) is composed of:

- o Architectural specification
- o Traceability to the domain model
- o Tradeoffs and rationale for the generic architecture(s)
- o Guidelines for using the generic architecture(s)

The term *generic architecture* is defined in the Generic Architecture Study of Quanrud as follows:

> A generic architecture provides a high level design for a family of related applications and a set of reusable components that are intended for use in those applications. The reusable components are designed to work together and should provide most of the code that would be included in a typical application. Actual applications are developed by adding application specific components and adapting the reusable components to meet the requirements of the application. Adaptation of a reusable component may take the form of modification, extension, use-as-is, or replacement. [QUA88]

A central feature of a generic architecture is that it provides both a design and an integrated set of reusable components.[QUA88] As a design for the family of systems in the application domain, a generic architecture defines the structure and interactions of the elements of the system. As a specification of potential reusable components, the generic architecture defines a specific context and interfaces that the components must satisfy. Components in a repository normally must be usable with designs yet to be determined. The generic architecture defines how the reusable components interact with each other and with custom application software so that they can be specifically designed to do so. [QUA88]

This unique dual role of the generic architecture leads to the following conclusion as to the level of detail that the architecture should be taken:

> The generic architecture should reflect a software decomposition to the level of detail necessary to identify all of the reusable software components and how they interact.

Within the context of DOD-STD-2167A [DOD88], this likely means that the generic architecture must be completely specified to the level of CSUs (Computer Software Units). This would require detail to the level to complete the Software Design Document (SDD). Within the structure of the generic architecture, they may be a mixture of reusable CSUs, CSCs (Computer Software Components), or even reusable CSCIs (Computer Software Configuration Items).

The development of a generic architecture for the domain employs a full systems engineering process. The domain model is reviewed and, if necessary, refined or rescoped based upon new knowledge.

The generic architecture representations should be object-oriented. While the definition of the generic architecture will likely be elaborated in a top-down fashion, the process of deriving the architecture involves the synthesis of the objects discovered in the domain model with those potential enabling component objects that were identified into a structure that meets all of the additional complementary requirements of the application domain. Traceability should be maintained to the domain model.

Multiple alternative object-oriented architecture(s) are postulated to meet the requirements presented by the domain modeling activity. Alternative analysis ensures proper consideration of key system design issues such as performance, reliability, fault tolerance or survivability and encourages convergence on an "optimal" architecture. Typical analyses performed include:

- o Timing and sizing analyses
- o Reliability/survivability analyses
- o Operations support analyses
- o System effectiveness analyses
- o Cost and cost effectiveness analyses

Sufficient tradeoffs and analyses are conducted to address acceptable system operational performance. The architectural analysis proceeds iteratively, by level of abstraction, to lower levels of detail. The results of these trade-off analyses are documented with the resulting generic architecture.

The architecture is documented using a combination of graphical and textual techniques that specify:

- o Decomposition of objects into subobjects
- o Object architecture and interaction (control and data) flows
- o Dynamic behavior of the architectural elements
- o Key external and internal interfaces
- o Physical configuration(s) and constraints

Since one single architecture may not be feasible to meet all of the adaptation requirements, multiple architectures may be defined with rules on when to use each. This may be required to accommodate projected future changes in the underlying hardware or software architectures.

The resulting generic architecture(s) defines the overall system "template" for future system instances in the domain and identifies specific reusable software components. In general, the development of generic architectures streamlines the development of new system instances in the domain. Preliminary and detailed design of a new system need only be an extension or customization of the generic architecture.

The guidelines for using the generic architecture(s) should address associated standards and policy issues. For example, will the generic architecture be mandated for future systems, or will it merely be used as a starting point? To what level of detail will the architecture become standard? Will key interfaces be standardized? What hardware and software standards will be applicable? How will changes to the generic architecture be controlled?

Even if the decision is made not to mandate a generic architecture or its internal interfaces, such a policy decision does not diminish the importance of this generic architecture step. Proper design of reusable components requires an analysis of the uses of the component and the environments in which they must integrate. A generic architecture motivates such analysis. Even when not mandated, a generic architecture provides both a starting point (i.e., a reusable architecture) for system instances, and a basis for analyzing the appropriateness, technical feasibility and economic advantages of investing in specific reusable component assets.

3.3.7 Validate the Generic Architecture(s)

This step results in a validated generic architecture or family of architectures. Additional outputs are the generic architecture validation products that include:

- o Architecture simulations
- o Prototypes
- o Partial implementations

Simulation of the system architecture may be effectively employed as an analysis tool and also to validate the architecture. Key architectural tradeoffs involving performance issues are often best accomplished using simulation.

However, validation of the generic architecture(s) is best accomplished through prototyping or other implementation. A full or partial implementation employs the generic architecture to meet a specific set of future or past set of requirements. The architectural and implementation risks should dictate the level of implementation necessary to validate the architecture.

The completion of the generic architecture represents another milestone at which to revisit the cost-benefits of the recommended approach. Sizing of the various software elements, both reusable and custom, yields a more accurate estimate of the both the costs of reuse and the potential savings. This supports a decision on whether to proceed with the implementation of the associated reusable software component assets, or the selection of ones which will provide the best cost-benefit ratio.

3.3.8 Develop Software Component Classification(s)

This step develops the component classifications that will be used to organize and retrieve software components.

The set of reusable components to be acquired and/or developed are identified in the generic architecture. The architecture validation effort will have selected those to be implemented based upon cost-benefit analysis. These components are organized and classified to expedite their development and usage. The component classification scheme that is derived may be based upon:

This page intentionally left blank.

o Domain dictionary

o Domain model

o Adaptation analyses

o Generic architecture

o Previously existing classifications

The four most popular techniques for software component classification are *taxonomies*, *faceted classifications*, *semantic nets* and *clustered organizations*.

A taxonomy is a hierarchically graded arrangement of components in library. A hierarchical taxonomy provides a convenient way to group and arrange components according to an ordered series of concerns (represented by levels in the hierarchy), much like a table of contents. A component's classification is determined by its placement in the taxonomy. Taxonomies are easy to construct and easy to use, but are limited in their flexibility.

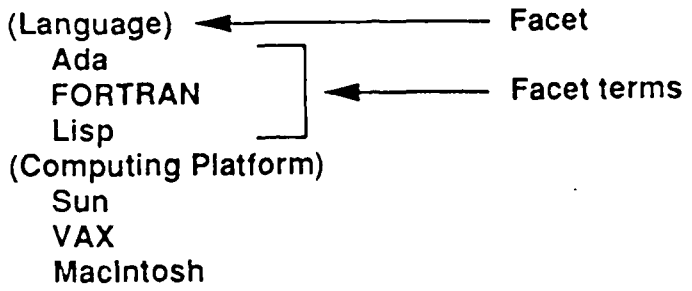
A faceted classification scheme identifies a set of facets, or categories, applicable to components within a library. These facets characterize the component by identifying distinguishing attributes (in the form of facet terms) that can be used to classify, search, understand, evaluate and select components. A component's classification is determined by the aggregate, or synthesis, of its attributes for all facets. Faceted classification schemes are more flexible than taxonomies and are easier to modify and expand. Figure 3.3.8-1 illustrates equivalent faceted and taxonometric classifications.

Semantic nets of terms or reusable components are often useful in classification and retrieval. Semantic nets are graphs whose nodes represent objects and whose arc represent relations between those objects. A semantic net can be developed as part of the domain dictionary to describe the conceptual closeness [PRI87] of domain terms. A semantic net may be derived directly from the domain model to express relationships between domain objects. Similarly, a semantic net may derived from the generic architecture to express the relationships of architectural objects.

Clustered organizations gather objects into clusters where the members of a cluster are more similar to each other than to the members of other clusters. Clustering is a mechanism that, like semantic nets, can be used to relate or associate terms or reusable components. A key advantage of clustered approaches is that associations can be accomplished automatically by a program based upon clustering rules, rather than being manually identified (as is the case with semantic nets). The clustering rules identify those criteria that define when "enough" similarity exists to be included in a cluster.

The particular classification scheme depends upon the viewpoint of the classifier. Multiple classifications may be developed to support different perspectives or purposes. For example, different classifications may be desired to deal with application domain (i.e., horizontal domain) objects and enabling component (i.e., horizontal domain) objects. Other classification schemes might specifically address a component's role in the generic architecture or its use of specific interfaces.

Faceted Classification



Hierarchical Taxonomy

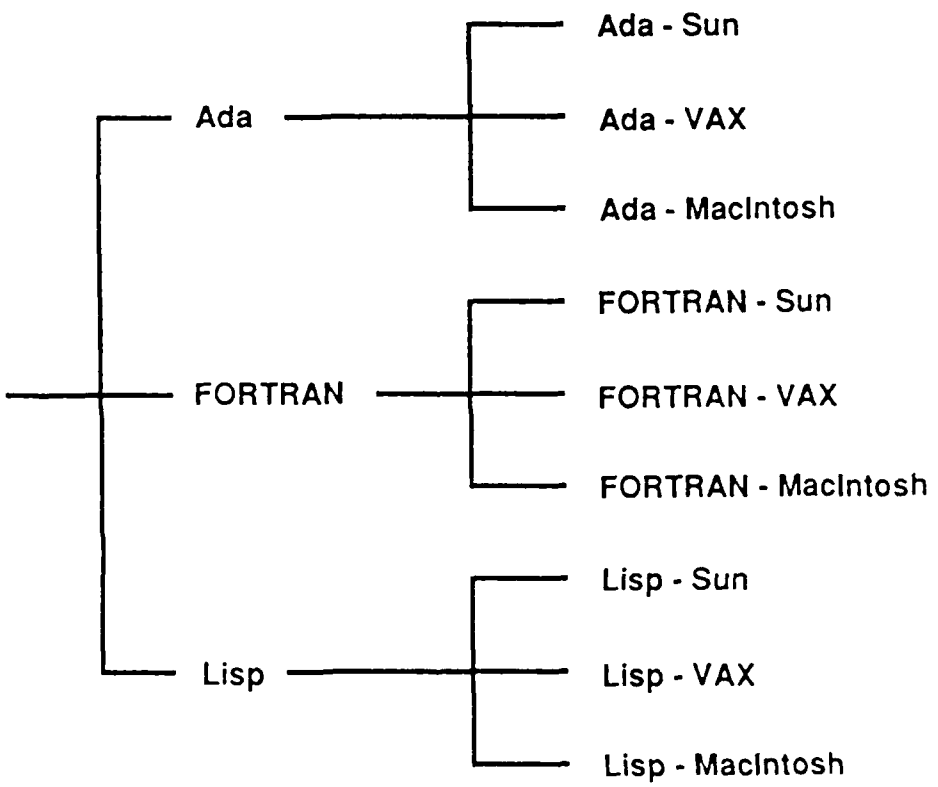


Figure 3.3.8-1. Classification Schemes.

3.3.9 Define Software Component Interfaces and Protocols

This step produces the interface and protocol specifications needed to completely specify the generic architecture(s) and allow the development of reusable software components that may be applied in the context of that architecture. The outputs include:

- o Interface specifications
- o Guidelines for applying the interface definitions

The interfaces that are documented are those interfaces to reusable software components or component sets and also key external and internal interfaces necessary for effectively applying the generic architecture.

The detailed design of the reusable components identified in the generic architecture effectively begins by defining the component interfaces and by defining their interaction protocols. The design of the component's interface employs the appropriate adaptation mechanisms, as identified in the earlier adaptation analyses for the domain model objects and for the enabling component objects.

Each interface definition should include the following information:

- o Purpose of the interface
- o Data formats and flows
- o Interaction protocols
- o Fault handling mechanisms
- o Interface layers (if applicable)
- o Physical, electrical or mechanical characteristics (if applicable)
- o Ada interface specifications (if applicable)
- o Scenarios of usage

The guidelines for applying the interface specifications must address policy issues associated with applying and controlling the interface definitions. Here, the primary issue is the degree of standardization that will be imposed.

3.3.10 Build and Catalog Reusable Software Components

This step develops, documents and classifies the software components that have been identified for the domain. Each software component consists of:

- o Ada source code
- o Documentation (it is recommended that this minimally comply with the DOD-STD-2167A requirement for Software Development Files (SDF))
- o Component traceability to the domain model
- o Component traceability to the generic architecture(s)

o Adaptation mechanism or constructor (if applicable)

o Tradeoffs and rationale for the component

o Description or placement in the classification scheme(s)

o Guide to using the component

Development of components is accomplished as a precursor activity, or as an accelerated development parallel to the first system instance development, as discussed earlier. It is important that the domain model and the generic architecture be maintained to be *consistent with the actual components* that are implemented.

Cataloging of the components is based upon the classification scheme(s) developed previously and is kept consistent with the baselines of the domain model.

The guide to using the component provides usage instructions and also addresses policy issues associated with data rights to the component, component baselining and configuration control. Risks associated with the component use are also identified and discussed.

3.3.11 Validate the Software Components

This step completes the process by providing a set of validated software components for use in the application domain. Other outputs include the software component validation products, such as:

o Test/certification plans and procedures

o Test cases, test data and test results

o Formal proofs of correctness

o Executable demonstrations

All components that are to be used in multiple system instances undergo rigorous testing and certification. The domain model objects form a specification basis for designing component test suites. These tests validate both these requirements and that the components adhere to the appropriate interfaces and protocols required for the generic architecture(s). Their ability to adapt based on previous adaptation analysis is also tested.

3.4 Resources Supporting Domain Analysis Activities

The personnel requirements for performing a domain analysis are demanding in both the breadth of skills required and the level of experience and capability that is needed. Because domain analysis is a systems engineering activity for a family of systems, the team that would be required is similar to that needed to perform a thorough systems engineering function.

The recommended team is composed of the following types of personnel:

o *Application domain expert(s)*, with knowledge of several previous systems in the domain and insight into the future requirements and

evolutions of the domain (ref. 2.6.2). This person minimally acts as a consultant to the domain analysis team, assisting in projecting future system requirements, defining the domain model and generic architecture, finding and analyzing existing systems, and reviewing the domain analysis work products.

- o *Systems engineer(s)*, performing the role of the domain analyst (ref. 2.6.3), doing the domain modeling, generic architecture and interface definition activities. Often this role may be filled by systems engineers that are also domain experts.

- o *Software and hardware specialists* to participate in the development of the generic architecture(s) and to advise on adaptations needed to accommodate underlying technology evolutions.

- o *Software engineers* to design and implement the reusable software components and to participate in prototyping or partial system implementations supporting validation of domain analysis work products (ref. 2.6.4).

- o *Software technologist* providing and evolving the tools (ref. 2.6.5) and techniques (ref. 2.6.1) applied for domain modeling, architecture derivation, systems analysis, component development and classification. This person is responsible for training personnel in the methods to be applied, installing and maintaining tools to support the methods and assessing conformance to standards and guidelines.

- o *Assurance engineer* to maintain configuration control of the domain analysis work products and to plan, manage and execute the validation activities of those work products.

- o *Program manager* manage the domain analysis activity and also to provide insight into programmatic plans and constraints and to address acquisition, legal, standards and policy issues (ref. 2.6.6).

While one person may, in fact, serve multiple roles, it should be noted that a team of less than three is unlikely, even for a small application domain. For large domains, the domain analysis team could easily be ten times this size.

3.5 Proposed Tools for Domain Analysis

Domain analysis has been defined as a systems engineering process, whose activities span a full development life cycle. In this section, we describe specific automated capabilities needed to support the proposed domain analysis methods, discuss how domain analysis tools, reuse libraries, and systems/software development environments might be related, and propose an approach to evolving domain analysis support from loosely coupled commercial-off-the-shelf (COTS) tools to a fully integrated systems engineering environment with specific support for domain analysis.

3.5.1 Automated Capabilities for Domain Analysis

An analysis of the activities and products proposed for domain analysis yielded the minimal toolset summarized in Table 3.5.1-1, and in the following paragraphs.

Table 3.5.1-1 Minimal Domain Analysis Toolset

- **Model the Domain**
  - **Scope the Domain Analysis**
    - Text and graphics editing tools
    - Cost modeling tools
    - Tracking and scheduling tools
    - Productivity assessment tools
  - **Perform Domain Commonality Analysis**
    - Data dictionary or database management system
      - Data management
      - Report generation tools
    - Language definition and processing tools (optional)
    - Object modeling tool(s)
      - Object specification tools
      - Object model analysis tools
      - Documentation generation tools
    - Text and graphics editing tools
    - Traceability tool or database management system
    - Configuration management tool
      - Version management
      - Revision/variation management
  - **Perform Domain Adaptation Analysis**
    - Text and graphics editing tools
    - Annotation and/or cross-reference tools
  - **Validate the Domain Model**
    - Text and graphics editing tools
    - Executable specification languages (optional)
    - Simulation tools
      - Constraint evaluation
      - Resource utilization
    - Cost modeling tools
- **Architect the Domain**
  - **Identify Enabling Component Base**
    - Reuse library
      - Browsing tool/query subsystem
      - Usage monitoring tool
    - Object modeling tool(s)
    - Text and graphics editing tools
  - **Develop Generic Architecture(s)**
    - Architecture design tools
    - Text and graphics editing tools
    - Traceability tool
    - Systems analysis tools
      - Timing and sizing analysis
      - Reliability analysis
      - Hardware-in-the-loop
    - Cost modeling tools
    - Configuration management tool
  - **Validate the Generic Architecture(s)**
    - Simulation tools
    - Prototyping tools
    - Implementation tools (see below)
    - Configuration management tool
    - Cost modeling tools

Table 3.5.1-1 Minimal Domain Analysis Toolset (continued)

Develop Component Assets
- **Develop Component Classification(s)**
  - Reuse library
    - Classification specification tool
  - Traceability tool
- **Define Component Interfaces and Protocols**
  - Text and graphics editing tools
  - Architecture design tools
  - Configuration management tool
- **Build and Catalog Reusable Components**
  - Implementation tools
    - Language-sensitive editing
    - Compilation
    - Linking/loading
    - Program library management
    - Static analysis/metrics tools
    - Instrumentation and dynamic analysis tools
    - Debuggers
  - Program generation tool (optional)
  - Text and graphics editing tools
  - Reuse library
    - Cataloging tool
  - Configuration management tool
  - Traceability tool
- **Validate the Components**
  - Testing tools
    - Test case/data generators
    - Test coverage analyzer
    - Testbed/test harness
  - Correctness proof tools (optional)
  - Text and graphics editing tools
  - Configuration management tool

Tools required during the domain scoping activity include support for the creation, maintenance and access of domain planning information such as domain history and rationale, scope of the domain and/or analysis activity, plan for the domain analysis effort, results of cost/benefit and risk analysis, etc. General purpose text and graphics editing tools could be used to document most of this information. Project management tools are needed to support development of the plan for domain development. These tools would be used throughout the domain analysis to track progress of the development against the plan. Cost modeling tools are needed to support analysis of cost/benefits for the domain. A database of information supporting the scoping activity, such as domain technology studies, identification of domain experts, existing systems documentation, etc. would be helpful.

During the domain commonality analysis activity, tools supporting the creation, maintenance and access of a domain dictionary and a domain model are required. Support for the domain dictionary could be provided through a data dictionary tool or a database management system. Definition of the domain model requires the support of one or more specification tools, ideally one which provides direct support for the recommended object-oriented modeling concepts. The object modeling tool(s) should support various static analyses, such as for completeness and consistency of the model, in addition to supporting creation and editing of the model documentation. Support for generation of documentation, such as conforming to the DoD-STD-2167A Data Item Descriptions should also be provided by the object modeling tool(s). A tool for defining and maintaining traceability to the domain plan is also needed (this could be a database manager). A traditional configuration management tool is needed to manage multiple versions of the model as it evolves over time. Formal domain languages could also be defined as part of the commonality analysis. These languages would be used to document the domain model and perhaps other domain analysis products as well. If a formal domain language is to be used, tools for defining languages, analyzing products written in those languages, and to generate products written in those languages will be needed.

Object modeling tool(s) which provide a capability to attach annotations to objects can be used to describe object differences identified during the domain adaptation analysis. Alternatively, adaptation requirements could be created and managed separately, and cross-reference tools used to document the links to objects which must be adapted. Annotations and/or cross-referencing tools could also be used to document "supplementary" domain model requirements identified during the commonality analysis, such as performance or security characteristics. Guidelines for using the domain model are most likely documented using general-purpose text and graphics editing tools.

General-purpose text and graphics processing tools could be used to document domain operational scenarios during the domain validation activity. Simulation tools are required to appropriately evaluate the ability of the domain model to meet specific system needs. Domain language processing tools which support execution of the domain model directly could be used instead of general-purpose simulation tools. Cost modeling tools are again applied to support reassessment/rescoping of the domain analysis effort.

General-purpose text and graphics editing tools are used to document the domain technology forecast affecting the identification and selection of potential enabling components for architecting the domain. The primary tool supporting the identification of existing reusable assets is a component browser and/or query subsystem provided by the reuse library. An expert system supporting query formulation and component adaptation analysis based on domain requirements would provide the ultimate in automated support for this activity. A reuse library which supports usage auditing and reporting can assist in evaluating past usefulness of components in the same or related domains, and in monitoring actual reuse which occurs during system instance development. Since further

elaboration of the domain model may also be applied to identification of enabling components, the object modeling tool(s) may still be of use.

Generic domain architectures are documented using architecture design tools that support a combination of graphical and text techniques. Object-oriented design tools are recommended. Static and dynamic systems analysis tools which support measurement of key design characteristics such as timing and sizing, and system reliability should be applied as they would for any architecture effort. A configuration management tool is used to maintain the one or more alternative architectural baselines implementing the domain model. Traceability to the domain model is documented and maintained using the traceability tool or database. Guidelines for using the generic architecture(s) are most likely documented using general-purpose text and graphics editing tools.

Again, the class of tools needed for the generic architecture validation activity are the same class of tools that would be applied to validating systems architectures in a typical development. Simulation and/or prototyping tools may be applied to address outstanding architectural issues and tradeoffs, which may or may not be domain-specific. If a full or partial implementation is to be used to validate the architecture, then implementation tools such as those described for the component development activity are used at this time. Cost modeling tools are again applied as more accurate estimates of the size, degree of potential reusability, and other characteristics of reusable components and custom software in the system family are computed.

Development of software component classification(s) should be supported by tools provided by the reuse library system. Many classification techniques are possible (taxonomies, facets, semantic nets, clustering), and so a tool which flexibly supports the full range of techniques is recommended. A tool supporting traceability of the scheme to the domain dictionary, model, and/or generic architecture is needed. Alternatively, traceability of the components themselves to the domain dictionary, model, and/or generic architecture(s) might be used as a minimal automated classification capability.

Component interfaces and protocols are more detailed specifications of generic architecture components, and the same architecture design tools used for the earlier activity continue to be used. Management of potentially large numbers of component variants must be addressed by the configuration management tool chosen. General-purpose text and graphics editing tools can be used to document the guidelines for applying the interface specifications.

According to the proposed domain analysis method, reusable components are object-oriented Ada program units. As such, building the components requires the support of at least a minimal Ada programming environment (compiler, library manager, debugger, etc.). Special static analysis tools such as those which compute various program metrics could be used to evaluate component reusability. Program generation tools may be available which could support automatic creation of Ada code from the domain model or architecture specifications. In addition, tools for documenting and managing traceability of components to the generic architecture(s) and domain model, and configuration management of the components are needed. General-purpose text and graphics editing tools can be used to document component usage guidelines, tradeoffs and rationale, and adaptation mechanisms for each component (or group of related components). If a reuse library is employed, the component is entered into the library according to the classification scheme developed previously. Certain characteristics used in the classification might be automatically extracted or derived by a tool from the component source code.

Support for the component validation activity is provided by traditional program validation tools such as test data generators, statement coverage analyzers, test harnesses, etc. Word processing tools are used to document component test plans and procedures. Projects with high reliability requirements may include tools supporting proofs of correctness. Component test programs, data, results, etc. must be managed along with the components, and so a configuration management tool, traceability tool, and/or a reuse library are also needed to perform this activity.

Since domain analysis is an iterative process, evolution of the domain analysis products will occur over time. Tools to identify, assess and manage change (e.g., impact analysis tools) will be useful over the life of domain information. Also, change request and problem reporting tools are needed.

3.5.2 Relationship to Reuse Library and Systems Engineering Environment

The possible uses of a reuse library are not fully indicated in the discussion of domain analysis tools above. Ideally, the reuse library is the repository for all of the information produced and/or used during the domain analysis process, because all of the resulting products have a potential for reuse. Reuse of a particular item may occur during the same domain analysis in which the item was developed, during subsequent domain analyses, or during subsequent system developments. Due to the overlapping and intersecting nature of many domains, the products of the analysis of one domain may also be reused in the analysis of another domain.

The primary candidates for reuse are domain models, generic architectures and software components. A domain model may be used as the starting point for the requirements specification of a new system in the domain. A generic architecture may be used as the starting point for sibling generic architectures. The collection of software components from a given generic architecture may be used to directly construct new applications within the domain. To a lesser degree, individual domain model objects, architectural components and document fragments may be reused in new system implementations.

A minimal reuse library system provides the following capabilities supporting the reuse of domain analysis products:

- o Classification and cataloging
- o Searching and browsing
- o Extraction and adaptation
- o Usage monitoring and reporting
- o Creating and maintaining relationships among objects/components/data

The reuse library system may manage several logical or physical repositories, which may be distributed according to security, geographical, policy, sizing and other constraints. Domain information may be stored and managed directly by the reuse library, or the reuse library may provide mechanisms to access information stored in external repositories (such as a configuration management system or program library).

It is obvious that the tools required to support domain analysis subsume the set of tools one would find in most modern systems/software engineering environments. Since the overall goal is to reuse the domain analysis products in new system developments (and

also to update the domain analysis products as new systems are developed), we recommend that same set of tools and same information repositories be used for both the domain analysis and subsequent system developments. This is illustrated in Figure 3.5.2-1, the Ultimate Integrated Domain and Systems Engineering Environment. However, practical considerations and exciting new tool technologies may lead to different environmental configurations, as illustrated in Figures 3.5.2-2 through 3.5.2-4, and described below.

Since the domain analysis and subsequent system developments may be performed by different organizations, and environments are likely to evolve over the life of a domain, the environment used to perform the original domain analysis and the environment(s) used to implement new system instances may not be the same. Figure 3.5.2-2 shows independent domain and systems development environments. Within the domain development environment, we may find separate independent toolsets/environments to handle each of the domain/architecture modeling, component development, and reuse library management. The reason for this is that existing toolsets often manage private information repositories specific to that toolset. Special import/export tools are probably required to move and/or access information across toolsets.

Figure 3.5.2-3 illustrates a scenario where maintained consistency between the domain and systems development environments permits sharing of a common toolset for both. One thing that is not obvious from the tools analysis above is what set of tools or capabilities are required to uniquely support domain analysis. Our analysis indicates that these would include tools with specific knowledge of particular application domains (for example, a command and control system prototyping tool, or a navigation systems analysis tool). These tools are typically used along with more general-purpose system development tools. Tools which might be considered unique to domain analysis are domain/component classification and browsing/retrieval capabilities. Because such tools are more applicable to reuse in general (e.g., a facet definition tool need not be domain-specific), these tools are allocated to the reuse library. Again, the various toolsets are likely to manage their own private information repositories.

The most usable and effective domain/systems engineering tools may be those that are tuned to the requirements of particular domains. These are most likely to be custom expert systems, such as might be able to interact with engineers in an informal domain-specific way, while internally managing and/or deriving more formal domain representations. While custom tools could be built to support each domain of interest, the technology exists to automatically generate some of these tools (e.g., specification editors and analysis tools for domain-oriented languages and models), and potentially to generate whole domain-specific development environments. Figure 3.5.2-4 illustrates these concepts. It is implied that the ultimate integrated domain and systems engineering environment would include domain-specific analysis capabilities.

3.5.3 Evolution of Domain Analysis Tool Support

We recommend an evolutionary approach to the acquisition and/or development of a domain analysis environment in which the degrees of integration and domain-specific support increase over time.

The primary near-term strategy (1-2 years) is to acquire a set of commercial-off-the-shelf (COTS) environments/tools covering as much of the minimal set of domain analysis automation capabilities listed in section 3.5.1 as possible. This implementation will yield a loose collection of generally domain-independent tools and databases. Some degree of integration of the tools can be accomplished by providing a common reuse library and facilities to import/export information to/from these tools to/from the reuse library.

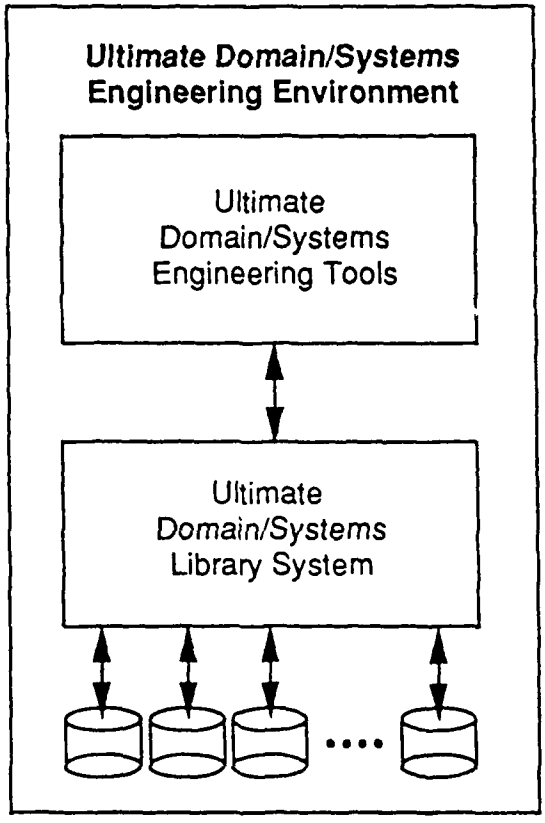


Figure 3.5.2-1 Ultimate Integrated Domain and Systems Engineering Environment

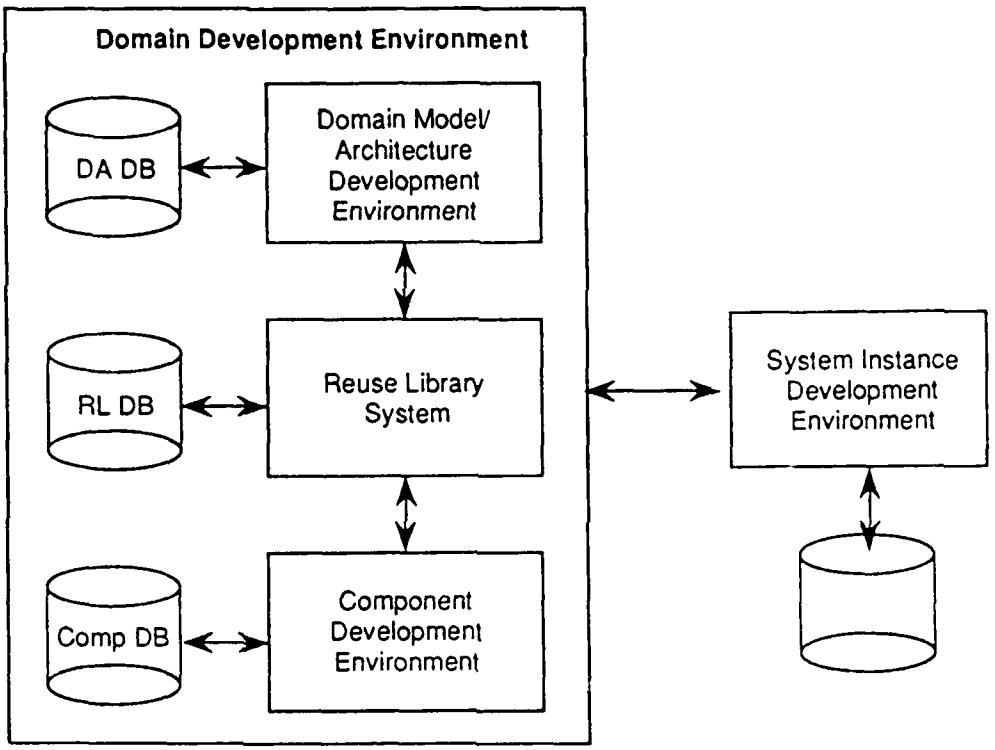


Figure 3.5.2-2 Separate Domain and Systems Engineering Environments

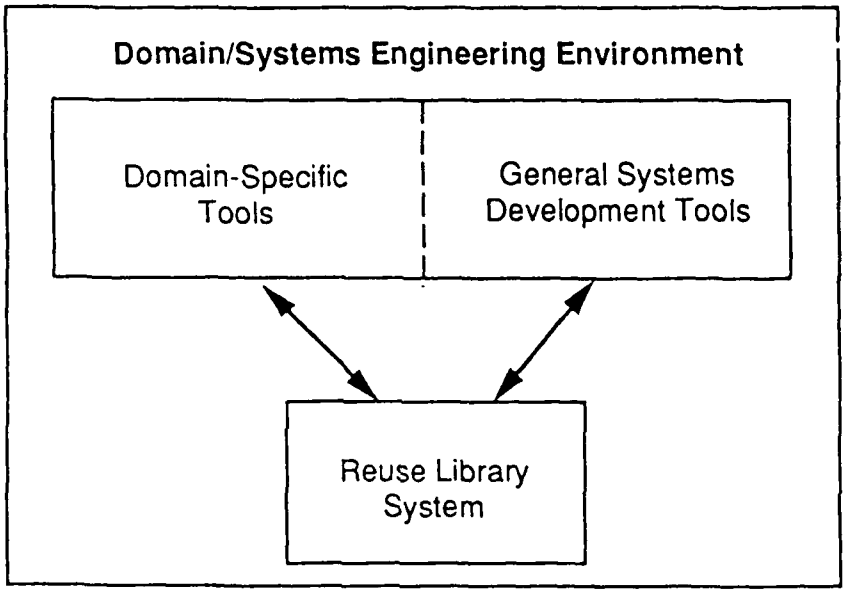


Figure 3.5.2-3 Integrated Engineering Environment with Domain-Specific Tools

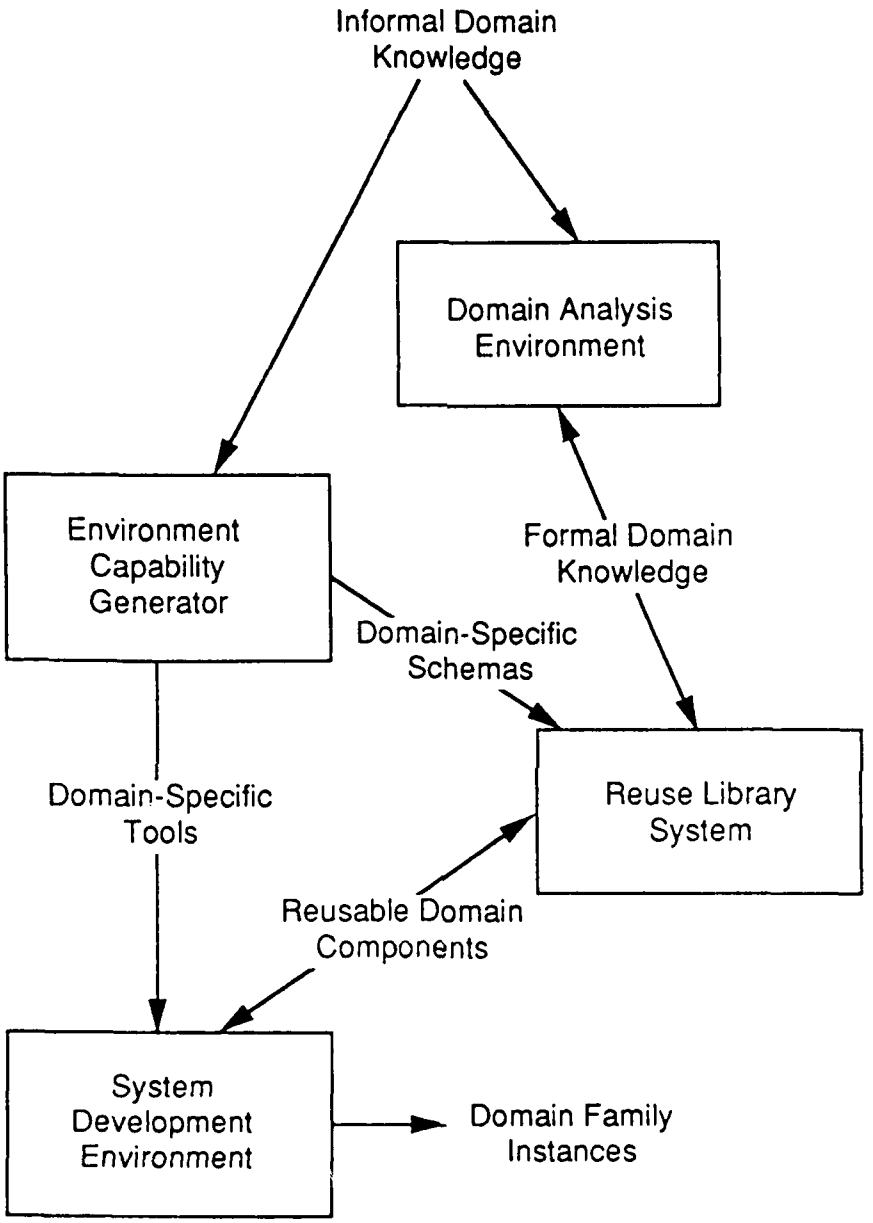


Figure 3.5.2-4 Application of Environment Generation Technology

Selection of a set of tools supporting a consistent development methodology (we recommend object-oriented) increases the ability to meaningfully share information across tools. Also, because the proposed method for domain analysis is object-oriented, it is compatible with existing and emerging expert systems approaches, tools, and representations. We recommend a study in the near-term to formalize an approach to integrating expert systems technology into the domain analysis methods and toolset.

The primary mid-term strategy (2-5 years) is to increase the degree of integration among the domain analysis tools through adoption of a common environment framework. An environment framework is the underlying hardware and software infrastructure that implements and manages domain/system information, computing resources, inter-tool communications, tool execution, access control, security and the user interface. Tool integration is accomplished through:

- o Information integration, the ability to share data and its meaning. Information integration is the most critical requirement to accomplishing tool interoperability. A common information repository and the definition of information interchange protocols are the primary mechanisms for achieving information integration.

- o Control integration, the ability to manage the execution of tools. Control integration can improve the effectiveness of tools through better tool cooperation and distribution of processing tasks.

- o User interface integration, the ability to provide a common user interface across all tools. This includes presenting a common "look and feel" to the users of all tools as well as providing a common set of underlying user interface facilities for implementing user-tool interaction.

- o Method integration, the ability to control the usage of tools to conform to a particular development method. This also includes facilities common to all development activities, such as configuration management.

Cooperating tools which are tightly coupled to the framework are acquired and/or developed over time to replace existing loosely coupled tools. Naturally, the reuse library evolves to become the common information repository.

The results of the expert systems analysis are applied to evolution of the domain analysis tools and environment. Also, tools and techniques for acquiring, analyzing and applying knowledge of domains and the domain development process itself are rapidly maturing, and may be suitable for integration into the domain analysis at the latter end or just beyond this time frame.

Investigation into knowledge acquisition techniques is particularly of interest because a domain analysis involves extensive use of expert knowledge in the application domain of interest. According to Parsaye, "to automate the process of knowledge acquisition and validation, we need tools that:

- o Help experts without a knowledge engineer capture their own expertise

- o Allow knowledge engineers to capture knowledge more effectively in very complex applications

- o Provide methods of automatic induction that capture knowledge usually unobtainable without computer assistance

o Provide facilities for managing a data base of test cases and evaluating them to measure the quality of the captured knowledge" [PAR88]

There are several approaches to knowledge acquisition, each having advantages and disadvantages with respect to issues such as:

o The degree of automation possible

o The amount and type of data required as input or provided as output

o The degree of user intervention and the roles of the users

o Validation of the knowledge

As well as expert knowledge in the application domain, knowledge of how to perform the domain analysis process itself could be used to guide application engineers, reducing the need for domain analysis experts during application development.

Existing domain analysis approaches have resulted in the development of multiple languages for defining domains and other products of the domain analysis process. Standards will be needed in order to reuse information across domains, and to support the exchange of information between different tools in the environment. Natural language processing tools are desired to avoid forcing the user to develop, learn and use complex domain definition languages. Such tools would also be useful in extracting domain information from documentation for existing applications.

The long-term goal (5 years and beyond) is to complete and continue to enhance the development of a fully integrated domain/systems engineering environment and repository which includes automated domain-specific expertise.

3.6 Addressing the Issues and Risks of Domain Analysis

Given the relatively immature state of reuse technology, there are many outstanding issues and risks involved with domain analysis. The following areas of concern have been identified:

o Impact of Development Methodology

o Impact of Development Languages

o Impact of Development Tools

o Impact of Development Personnel

o Impact of Development Organization and Policy

o Impact of Application Systems

o Evaluation and Validation Concerns

o Domain Maintenance and Evolution

o Domain Analysis Techniques

Each of these issues are discussed in the following sections.

3.6.1 Impact of Development Methods

This section addresses questions applicable to the impact of development methods on domain analysis. The following issues must be considered when doing an impact analysis:

- o What are the impacts of using a particular method? Does use of a *particular domain analysis method* preclude reuse with others?
- o Which methods best contribute to achieving reuse goals?
- o Can reusable components be repackaged to work with any method?
- o Can reusable components simply be collected and used in an ad hoc fashion?

In order to develop reusable components, a design method must be used. If the design method selected for developing reusable components is not compatible with the design method(s) being used to develop application systems, then the reusability of the components may be severely limited. Risks of this type can be reduced by reaching an agreement on the family of design methods (e.g., object-oriented methods, structured design methods) that will be used in both developing the reusable components as well as developing the application systems which intend to reuse the components. This risk can also be reduced by developing a generic architecture. The generic architecture provides a common framework for both the reusable components and the application systems thus going a step further to reduce the likelihood of incompatibilities due to design method.

At the current time, there is no incontrovertible proof that any of the design methods produces more reusable component than any other. However, the current thinking points toward the object-oriented design methods as producing components which are more reusable than other methods. One of the reasons for this perception is that object-oriented methods identify components which are closer to real-world objects, thus reducing the conceptual gap between the problem space and the solution space. Another reason for the perception that object-oriented design methods tend to produce more reusable components is that these methods require that external component interfaces are explicitly defined. A third reason that object-oriented is a current favorite is that object-oriented design is highly repeatable. A design method is considered repeatable if different people can come up similar designs for the same problem by consistently applying the method. Repeatability of a design method reduces the method risk since the application system designer is more likely to identify a similar component to the one identified during domain analysis and developed as a reusable component.

It is considered feasible to repackage reusable components to work with another similar method. For example, it would a relatively small impact to repackage a component designed with Hatley's Real Time Extensions to Structured Analysis/Structured Design (SASD) to work with an application system designed with Ward-Mellor's variation of SASD. However, it would be a much larger impact to redesign a component developed with SASD to be an object-oriented component. If fact, it might not even be technically feasible since object-oriented components are objects while SASD components are functions. It is also important to keep in mind when assessing this kind of impact that components are not just code. They include documentation and tests as well as code and the impact of changing everything must be considered.

In the past, libraries of "reusable" components have been collected and used in an ad hoc fashion. However, the reuse of these components have been limited since these libraries are often not supervised or maintained in the required manner. As a result, the

reusable components often do not work as indicated due to poor documentation or limited validation or both. These libraries have instilled a caveat emptor reaction in every software engineer which has a derogatory effect on well-engineered, well-maintained reuse libraries. As a result, an ad hoc approach to collecting reusable components may have a very negative impact not only on a project but also on the engineers.

3.6.2 Impact of Development Tools

This section addresses questions applicable to the impact of development tools on domain analysis. The following issues should be addressed:

- o What impact do tools have on classification schemes and information structuring?
- o What impact do tools have on the amount of information stored and reused?
- o How can we avoid locking in to language processing tools?
- o What impact does the effectiveness of the tools have on the desire to reuse?

Depending on the library system (e.g., database, configuration management tool, reuse library system) selected to store reusable components may have a big impact on classification schemes and information structuring. The tool may severely restrict the available types of classification methods (e.g., faceted) which may be used to classify the components during domain analysis. As a result, the domain analyzer should be aware of the limitations imposed by the selected library system so that the results of the domain analysis will be useful.

The selected library system may also have an impact on the amount of information stored and reused. Some library systems are capable of only storing source code while others can handle the storage of documentation and tests as well. As a result, during domain analysis, a practical way of storing component documentation in an easily accessible fashion must be addressed. Potential reuse is limited if documentation and tests are not as readily available as the code.

Another issue that needs to be considered during domain analysis are the risks involved with locking into a particular set of language processing tools to develop the reusable components thereby forcing any potential reuser to use the same set of tools in developing an application. One way to avoid locking onto a particular set of language processing tools is to select a language such as Ada. Reusable components can be developed in Ada, e.g., Booch's reusable components [BOO87], which are highly portable since Ada is a standardized language and all Ada compilers must be certified. If care is not taken during domain analysis to address portability issues, the resulting reusable components may be limited to use within applications which are developed using the same hardware, operating system, and compiler as used during the development of the reusable components.

The effectiveness that reuse tools have on the desire to reuse needs to be considered during domain analysis to insure that the resulting reuse system (components as well as reuse tools) is highly effective. Tools that are slow, unreliable, user unfriendly, or minimally functional will have a negative effect on the engineers desire to reuse software. As a result, the reuse tools should be analyzed and problems identified during domain

analysis. Identified problems should be addressed prior to release of the reuse system to the organization.

3.6.3 Impact of Development Languages

This section addresses questions applicable to the impact of development languages on domain analysis. The following issues need to be addressed during domain analysis:

- o How do selected languages support or preclude reuse?
- o How does the component implementation language relate to the domain specification language?
- o What are the impacts of using Ada?

Languages selected for use during domain analysis, component development, and application development may have impacts on the domain analysis and on the resulting components. Selected languages may support reuse in several ways. For example, the language may incorporate certain features, e.g., Ada packages, which provide support for representing reusable components better than a language which does not have such a feature. Another way in which a language may support reuse is through its portability. If components developed in a particular language are easier to port from one set of hardware, operating system and compiler to another set of hardware, operating system and compiler, then that language provides better support for reuse. Finally, if multiple languages are being used for component development and for the application system, then the languages should provide mechanisms for calling or referencing programs in other languages. During domain analysis, a survey of languages which will probably be used by application developers is important to develop to select the appropriate language(s) for component development.

During domain analysis, the domain and components with the domain may be represented using a domain specification language. Ideally, components in any implementation language could then be automatically generated from the domain specification language. However, this capability is still considered a research topic even though it has been worked on for more than twenty years. Therefore, the components must be manually implemented in a computer language. Depending on the selected domain specification language and the implementation language, the implementation process may be greatly simplified or greatly impacted. For example, if the domain specification language is object-oriented and the components are implemented in Ada (or Classic-Ada) then the implementation is greatly simplified. Consideration should be given to correct selection of languages both for domain specification and component implementation during domain analysis.

Since Ada is the language mandated by the U.S. DoD, it is highly likely that it will be the component implementation language for any DoD software development will be Ada. Ada has many features which support reuse such as generics, packages, and exception handling. Since it is a standard language and its compilers must be certified, it is possible to develop reusable components which are highly portable in Ada. Ada also has a language feature (pragma interface) which allows Ada programs to work with programs written in other languages. Any object-oriented specification language will work well with Ada. The guidelines on use of object-oriented methods to do domain analysis provided earlier in this section work well with Ada. However, Ada does not support inheritance and dynamic binding which have been shown to enhance reusability. Consideration should be given to using Classic-Ada which adds inheritance and dynamic binding capabilities to the Ada language. The Classic-Ada processor then translates the Classic-Ada code into Ada

code. No matter what language is chosen for implementing components, the risks and impacts associated with that selection should be analyzed during domain analysis.

3.6.4 Impact on Development Personnel

This section addresses questions applicable to the impact of domain analysis on development personnel. The following issues need to be addressed during domain analysis:

- o What is the appropriate division of effort between application experts and software engineering experts?
- o What new roles and knowledge are required for domain analysis?

During a domain analysis, effort must be divided between application experts and software engineering experts. If more time is spent by application experts, then the application model is higher quality and more functional. If more time is spent by software engineering experts then the reusability of the components is greater. The manager of the domain analysis process needs to recognize the potential impact of trading off time between these two groups of experts.

In this report, new engineering roles have been identified. These roles are required to effectively perform a domain analysis. Since there are few engineers with any experience at all in domain analysis, a learning curve should be anticipated in planning a domain analysis. Currently there is little or no training or education available in domain analysis either since it is a relatively new technology. As a result, the learning curve can only be shortened by including people with domain analysis experience on the team to guide the inexperienced engineers. Alternatively, time for experimentation and finding a good approach to accomplishing the domain analysis should be included in the plan.

3.6.5 Impact on Development Organization and Policy

This section addresses questions applicable to the impact of development organization and of policy on domain analysis. The following issues need to be addressed during domain analysis:

- o What is the impact of domain analysis on an organization?
- o What is the policy or procedural impacts associated with domain analysis?

Performing a domain analysis may cause organizational changes. The three approaches which have been identified [SEI88b] result in (1) a project-specific group, (2) a project-autonomous group, or (3) a multi-project group being formed to do the domain analysis. Each organizational structure has its pros and cons. For example, the project specific group will ensure reusability for its project but may not take other project needs into consideration. A project-autonomous group may produce a very effective domain analysis which is not 100% usable by any project. The project-autonomous group tends to look at all current and future needs from a theoretical position potentially providing operations and generality not needed currently. The multi-project group has to try to satisfy all project needs which may be inherently contradictory. Not only is the domain analysis group itself an impact to the organization but the organization must change to incorporate the results of the domain analysis into existing projects. Each project or organization may have an individual or a group responsible for insuring that reuse happens

on all projects to the maximum degree. Further, a group needs to be established to maintain the reuse library and to add or enhance components.

Once a domain analysis is complete, an organization needs to create a policy, a guideline, or procedure(s) to insure that the investment in the domain analysis pays off. The policy should require that reusable components be incorporated whenever possible into application project designs. Furthermore, the policy should deal with the case where there is an existing reusable component which doesn't quite meet the needs of the application project. In this case, the policy should indicate who is responsible for changing the reusable component to meet the projects needs. The policy should also address responsibility for maintenance of the components. Finally, the policy should address the responsibility of the application projects to identify and supply new reusable components to the reuse library.

3.6.6 Impact of Application Domain

This section addresses questions applicable to the impact of the application domain on domain analysis. The following issues need to be addressed during domain analysis:

- o Can domain analysis be effectively applied to the application domain?
- o Does the domain analysis process differ based on the kind of application?
- o Can application-specific components be repackaged to work in new application domains?

Before beginning a domain analysis, a determination must be made of the usefulness of doing a domain analysis for the particular the application domain. Different factors must be considered in doing this analysis including maturity of the application domain, availability of existing systems, potential degree of reusability in future systems, portability, and commonality among existing systems. In particular, embedded, real-time systems have been identified as an application area with limited potential reuse due to problems such as unique sensors and interfaces in every system, hardware dependencies related to timing considerations, and unique requirements for every system. However, if one limits the application area of real-time embedded systems to a particular kind of system such as fighter aircraft navigation systems then there may be more potential for reuse. Part of this determination of usefulness should be a cost/benefit analysis.

Experienced system engineers recognize that system analysis is done differently for every system. Even though a common strategy, e.g., object-oriented, may be used for every system, the techniques and processes are generally customized to suit the particular application being developed. One obvious reason to customize is different quality requirements, e.g., one system may require a high degree of reliability while another may require ease of maintainability. Different techniques may be required to insure that these quality goals are met. Domain analysis, a generalization of system analysis, is likely to follow a similar pattern of process and technique customization for each domain. The manager and engineers doing the domain analysis should consider the use of alternative techniques when necessary to get their jobs done. However, the impact of the different technique on the effort involved, on the quality of the product, and on the implementation of reusable components should be considered before applying it.

As our storehouses of application-specific components grow, new domain analyses may be able to reuse the results of previous domain analyses by repackaging or otherwise adapting existing components to work in their domain. When planning a domain analysis,

consideration should be given to existing domain analyses which may reduce the amount of effort required to do the new domain analysis.

3.6.7 Evaluation and Validation Concerns

This section addresses questions applicable to evaluation and validation concerns for domain analysis. The following issues need to be addressed during domain analysis:

- o How can the results of the domain analysis be validated?
- o What are appropriate measures of "goodness"?
- o Can domain analyses be evaluated in progress?

The most effective validation of a domain analysis is a high degree of reuse of identified components in an application system. However, the length of time necessary to accomplish this total validation is really too long to provide feed back to the domain analysis team. If the domain analysis is being done in conjunction with application project(s) then partial results from the domain analysis may be validated sooner by the application project(s). The manager of the domain analysis may instead choose to set up a mock application project to serve as a test bed for the domain analysis so that faster feed back may be accomplished.

Just as with any other form of systems and software engineering, metrics may be developed to measure the goodness of the domain analyses. At this point, there are no specific metrics identified for domain analyses. Research needs to be done in this area.

Verifying a domain analysis to insure that there is consistency between problem statement and the generic architecture and between the generic architecture and the reusable components is necessary. Reviews and walkthroughs should be used to verify the domain analysis in the same way that software developments are verified. Traceability should be used to insure coverage of requirements and to support impact analysis. Managers should include reviews and walkthroughs in their plans.

3.6.8 Domain Maintenance and Evolution

This section addresses questions applicable to the maintenance and evolution of domain analysis products. The following issues need to be addressed during domain analysis:

- o What kinds of changes augment the results of a previous domain analysis?
- o What kinds of changes invalidate the results of a previous domain analysis?
- o How does domain impact analysis differ from "normal" impact analysis?
- o What can be done to minimize domain change impacts?

Planning for change is the best way to mitigate impacts due to change. Any domain analysis, like any kind of system/software engineering, will go through many changes due to changing requirements, improved hardware technology, changes in interfacing systems, improvements in architectural design, and so forth. Some of these changes will augment the results of previous domain analysis while others may invalidate parts of previous

domain analyses. Determination of the degree of impact can be accomplished using impact analysis techniques similar to those used in analyzing a potential change to any system. However, with a domain analysis, impacts to existing application systems which have incorporated the results of the domain analysis in the form of architecture and/or components have to be taken into account. Changing the domain analysis itself may be a relatively minor impact with respect to the potential changes to application systems which have been based on the impact analysis. Managers and engineers working on domain analyses should try to identify areas of the domain which are likely to change. Then the domain analysts should insure that the generic architecture and components are defined to localize the potential impacts of the likely changes. Applying this kind of strategy (namely Dave Parnas' information hiding principle) will tend to minimize impacts due to change in the long term.

3.6.9 Domain Analysis Techniques

This section addresses issues and risks associated with the techniques used to perform a domain analysis. The following issues need to be addressed during domain analysis:

- o How should domains be partitioned and combined?
- o How are systems which comprise several domains handled?
- o How can domain boundaries be effectively handled?
- o How are domain overlaps handled?
- o Can domain analysis be accomplished incrementally?

When starting a domain analysis, a plan should be developed which defines the strategy to be used in approaching the domain analysis. One part of this plan should address the scoping of the domain, including partitioning and combination of domains, overlapping domains, and resulting in the clear definition of the domain boundary. An approach to incrementally accomplish the domain analysis is strongly recommended since complete products will be released sooner, and feedback can be obtained sooner which can then be used to improve the later increments of the domain analysis. Risks can be minimized by clearly scoping the boundary of the domain and by using an incremental approach.

4.0 SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

The primary objective of this effort was to develop an approach to make domain analysis practical and effective for the development of reusable software and the reuse of that software in new application systems. The main focus was domain analysis within the context of DoD software development; more specifically, the interest is in embedded Ada software for Army applications. To meet this objective, we developed a preliminary, yet comprehensive, guidelines for performing state-of-the-art domain analyses. The guidelines assist software developers and acquisition personnel in applying domain analysis to achieve reuse goals. The guidelines also suggest areas where future automation efforts might be directed.

Based on a survey of existing approaches, alternative approaches to domain analysis for Army Ada applications were analyzed, and the most promising selected for further development. A consistent, cohesive, and complete methodology for domain analysis which addresses the major issues was then developed.

Automated capabilities which support the domain analysis process were proposed. These capabilities address the application of existing tools to domain analysis, as well as future tool developments. This effort identifies and addresses the key technical areas which affect the automation of domain analysis. These areas include knowledge acquisition and knowledge-based guidance, domain languages and language-based processing, information models and data storage and retrieval, and tool and environment integration. Ideally, automation mechanizes methodology (as opposed to defining methods based on available tools). As such, the definition of domain analysis methods was completed before proposing tools to support the method.

Our results are presented in Section 3, Guidelines for Conducting a Domain Analysis, which describes a proposed methodology for domain analysis and identifies supporting tools.

Another objective of this effort was to reinforce the importance of domain analysis as part of a software reuse program. A particular goal was to assist in avoiding the potential negative effects of ad hoc or otherwise "bad" domain analyses. However, rather than discouraging attempts to use domain analysis, the goal was to encourage the experimentation and feedback necessary to advance in this new but critical technology area.

To address this objective, the developed guidelines address risk areas associated with doing a domain analysis. As with all emerging technology, any attempt to apply domain analysis presents many issues and risks. Although most of them are technical there are sociological and economic problems to be dealt with as well. This report begins to identify the critical issues and risks associated with the domain analysis process and postulate alternatives for resolving those issues and dealing with those risks. Risk areas addressed include the impacts of methods, tools, languages, personnel, organizational structure, policy, and application domain on the domain analysis process. Other important issues which should be addressed are evaluation and validation of domain specifications and maintenance of those specifications.

Our results are presented in Section 3, Guidelines for Conducting a Domain Analysis, which addresses dealing with issues and risks of domain analysis.

Another objective of this effort was to summarize and coalesce the varied and dispersed information on domain analysis into a single source of reference. Existing and emerging methods, languages, tools and techniques that have been or could be applied to

the domain analysis process were examined, and a snapshot of research results is provided.

The initial work performed under this effort was a survey of existing and emerging methods and tools for performing a domain analysis and applying its results. In order to assess and compare these various approaches, a set of evaluation criteria for domain analysis methods and tools is developed first. The criteria address seven broad areas:

- o Strategies and paradigms for domain analysis
- o Process models for domain analysis (including relationships to the overall process of software reuse)
- o Inputs to domain analysis activities
- o Methods for performing domain analysis activities
- o Products of domain analysis activities
- o Personnel supporting domain analysis activities
- o Tools supporting domain analysis activities

Section 2, Survey of Domain Analysis Approaches, summarizes the results of a survey of domain analysis research and case histories. In addition, Appendix C provides synopses of recent research efforts into domain analysis which were used in developing the main document.

As one of the first significant and comprehensive research efforts in domain analysis, this report identifies where tool prototyping, methods experimentation, and additional technical research are needed. In the following paragraphs, our major conclusions and recommendations are summarized.

Our primary conclusion is that domain analysis, when done right, is a significant undertaking yet with a significant benefit.

The guidelines for conducting a domain analysis are based on the following definition of domain analysis:

> Domain analysis is the systems engineering of a family of systems in an application domain through development and application of a reuse library.

The domain analysis must be focused to address a *family of systems* in a particular application domain. The more definitive these family of systems can be defined, the more focused and precise the domain analysis activity can be.

Domain analysis is, in fact, a *systems engineering* activity. It involves understanding and specifying the requirements for the family of systems (in the form of a *domain model*). The domain analysis activity includes developing a design for the family of systems (in the form of *generic architecture(s)*). It must include in this process all of the tradeoff and analysis techniques that are needed for proper systems engineering of any one of these systems (e.g., performance analysis, reliability analysis).

Central to the concept of domain analysis is, of course, reusability. Thus, the end goal of a domain analysis activity is to develop a reuse library *asset* that will be used in the

implementation of system instances in the domain family. These assets in the library will include software components and generators, documentation, interface specifications, test plans, procedures and data. In addition, the domain model and the generic architecture(s) are themselves valuable reusable assets.

Based upon the experiences of a number of researchers (including [BIG88], [LIE88], and [QUA88]), and motivated by the end goal of producing Ada software, *object-oriented* is selected as the underlying paradigm for the modeling and architecting of the domain and for the resulting design and development approach for the software component set (see Figure 3.1-1).

Such a unified object-oriented domain analysis process offers several advantages:

- o Provides a single, consistent model that requires no "great mental leap" from analysis to design and increases traceability and maintainability
- o Matches the technical representation of the software system more closely to the conceptual view of the application
- o Provides a stable framework for analyzing the problem domain and for levying requirements
- o Supports implementations in Ada using abstract data types

Object-oriented development allows a natural transition from the real-world problem space to analysis to design to implementation. Real-world application entities are specification objects discovered by performing object-oriented analysis. Object-based analysis approaches thus serve to capture a model of reality. Such approaches are well suited to managing the complexity inherent in large, software-intensive systems.[BOO87]

We concluded that a three phased approach should be used for domain analysis:

- o *Model the domain.* Based upon knowledge of the domain and programmatic constraints, an object-oriented domain model is developed and validated. This model is specifically analyzed to denote the necessary adaptations needed by envisioned future systems in the domain.
- o *Architect the domain.* A generic object-oriented Ada architecture(s) is developed and validated for the family of systems, identifying those software components that are potentially reusable. Standard systems engineering disciplines are applied to ensure that the architecture is a feasible and satisfactory basis for developing future systems.
- o *Develop software component assets.* A set of reusable object-oriented Ada components is built and validated to comply with the interfaces and protocols required by the generic architecture(s). These components are cataloged into the reusable component library for the domain.

Furthermore, we concluded that there are basically two approaches to performing a domain analysis in conjunction with the overall software development process:

- o As a precursor activity to future system developments in the domain
- o As a parallel activity to a development of an instance of a system in the domain

Our guidelines introduce the concept of adaptation analysis, i.e., the identification of differences among application systems in the domain. Adaptation analysis is critical to deriving a generic architecture and component library that can adapt to future system needs. This analysis of required adaptation may be based upon mission, threat, domain, or system planning information. Additionally, trends observed from an analysis of changes or evolutions of previous systems may be helpful.

As a result of our research, we have identified future directions for research and development in domain analysis to take. Our recommendations address validation and application of the domain analysis guidelines, future research topics in domain analysis technology, and evolutionary development of a domain analysis environment.

As the next step, we recommend application of our guidelines to a small, well-defined domain to validate the guidelines and provide pragmatic feedback on the application of the proposed methods. Furthermore, before attempting to perform a significant domain analysis, we recommend that a complete guidelines and associated training should be developed. At this point, we feel that a significant domain analysis could be attempted.

Topics identified which require additional research include:

- o Cost model for domain analysis leading to the development of a cost/benefit analysis
- o Metrics for domain analyses
- o Effective methods for validating domain analyses
- o Reuse of domain analyses products
- o Multiple domain issues
- o Impact analyses on domain analyses products
- o Knowledge acquisition of domain and software engineering knowledge
- o Adaptation analysis methods

We recommend an evolutionary approach to the acquisition and/or development of a domain analysis environment in which the degrees of integration and domain-specific support increase over time.

The primary near-term strategy (1-2 years) is to acquire a set of commercial-off-the-shelf (COTS) environments/tools covering as much of the minimal set of domain analysis automation capabilities listed in section 3.5.1 as possible. This implementation will yield a loose collection of generally domain-independent tools and databases. Some degree of integration of the tools can be accomplished by providing a common reuse library and facilities to import/export information to/from these tools to/from the reuse library.

Selection of a set of tools supporting a consistent development methodology (we recommend object-oriented) increases the ability to meaningfully share information across tools. Also, because the proposed method for domain analysis is object-oriented, it is compatible with existing and emerging expert systems approaches, tools, and representations. We recommend a study in the near-term to formalize an approach to integrating expert systems technology into the domain analysis methods and toolset.

The primary mid-term strategy (2-5 years) is to increase the degree of integration among the domain analysis tools through adoption of a common environment framework. An environment framework is the underlying hardware and software infrastructure that implements and manages domain/system information, computing resources, inter-tool communications, tool execution, access control, security and the user interface. Tool integration is accomplished through:

- o Information integration, the ability to share data and its meaning. Information integration is the most critical requirement to accomplishing tool interoperability. A common information repository and the definition of information interchange protocols are the primary mechanisms for achieving information integration.

- o Control integration, the ability to manage the execution of tools. Control integration can improve the effectiveness of tools through better tool cooperation and distribution of processing tasks.

- o User interface integration, the ability to provide a common user interface across all tools. This includes presenting a common "look and feel" to the users of all tools as well as providing a common set of underlying user interface facilities for implementing user-tool interaction.

- o Method integration, the ability to control the usage of tools to conform to a particular development method. This also includes facilities common to all development activities, such as configuration management.

Cooperating tools which are tightly coupled to the framework are acquired and/or developed over time to replace existing loosely coupled tools. Naturally, the reuse library evolves to become the common information repository.

The results of the expert systems analysis are applied to evolution of the domain analysis tools and environment. Also, tools and techniques for acquiring, analyzing and applying knowledge of domains and the domain development process itself are rapidly maturing, and may be suitable for integration into the domain analysis at the latter end or just beyond this time frame.

The long-term goal (5 years and beyond) is to complete and continue to enhance the development of a fully integrated domain/systems engineering environment and repository which includes automated domain-specific expertise.

Appendix A

List of Acronyms

ACCS - Army Command and Control System

ADT - Abstract Data Type

AMPEE - Automated Missile Parts Engineering Expert

AP - Applications Processor

APSE - Ada Programming Support Environment

CAMP - Common Ada Missile Packages

CASE - Computer Aided Software Engineering

CECOM - Communications Electronics Command

CMP - Conceptual Modeling Platform

CMS - Code Management System

COTS - Commercial Off-the-Shelf

CSC - Computer Software Component

CSCI - Computer Software Configuration Item

CTA - Computer Technology Associates

DA - Domain Analysis

DARTS - Development Arts for Real-Time Systems

DBMS - Data Base Management System

DD - Data Dictionary

DE - Domain Engineering

DESIRE - DESIgn REcovery

DFD - Data Flow Diagram

DID - Data Item Description

DL - Domain Language

DoD - Department of Defense

EBNF - Extended Baccus-Naur Form

EFT - Electronic Funds Transfer

E-R - Entity-Relationship

ERD - Entity-Relationship Diagram

FORTRAN - Formula Translation

GSED - Generalized Software Engineering Design

GOOD - Goddard Object-Oriented Design

GSFC - Goddard Space Flight Center

IBM - International Business Machines

MCCR - Mission Critical Computer Resource

MoD - Model of the Domain

MoT - Model of the Reuse Task

NASA - National Aeronautics and Space Administration

NEC - Nippon Electic Company

OOD - Object-Oriented Design (OOD)

OOPSLA - Object-Oriented Programming Systems, Languages, and Applications

OORA - Object-Oriented Requirements Analysis

PES - Parallel Elaboration of Specifications

POCC - Payload Operations Control Center

RLF - Reusability Library Framework

RTE - Run-Time Environment

RTEWG - Reuse Tools and Environments Working Group

SADT - Structured Analysis and Design Technique

SASD - Structured Analysis/Structured Design

SDDD - Software Detailed Design Document

SDF - Software Development File

SE - Software Engineering

SED - Software Engineering Design

SEI - Software Engineering Institute

SIGAda - Special Interest Group on Ada

SRE - Software Reuse Environment

SRS - Software Requirements Specification

SSS - System/Segment Specification

STARS - Software Technology for Adaptable, Reliable Systems

STLDD - Software Top-Level Design Documnet

VHLL - Very High Level Language

Appendix B

References

[ABB86] Abbott, Russell J. *An Integrated Approach to Software Development*, John Wiley & Sons, New York, NY, 1986.

[ARA] Arango, Guillermo, Ira Baxter and Peter Freeman. "Maintenance and Porting of Software by Design Recovery," Univ. of California at Irvine.

[ARA87] Arango, Guillermo. "Domain Engineering for Mechanical Reuse," Univ. of California at Irvine.

[ARA88] Arango, Guillermo. "On the Process of Acquiring Reusable Program Abstractions," Univ. of California at Irvine.

[ARM83] *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, United States Department of Defense, February 17, 1983.

[AUE88] Auer, Ken and Sam Adams. "Domain Analysis: Object Oriented Methodologies and Techniques," Knowledge Systems Corp., position paper for domain analysis working group session at OOPSLA '88.

[BAI88] Bailin, Sidney, and John M. Moore. "A Software Reuse Environment," Computer Technology Associates, position paper for Reuse Tools and Environments Workshop, Bass Harbor, ME, June 1988.

[BAR85] Barstow, David R. "Domain-Specific Automatic Programming," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 11, November 1985, pp. 1321-1336.

[BAR86] Bartussek, Wolfram and David L. Parnas. "Using Assertions About Traces to Write Abstract Specifications for Software Modules," *Software Specification Techniques*, Narain Gehani and Andrew McGettrick (eds.), Addison-Wesley Publishing Company, Wokingham, England, 1986.

[BAX88] Baxter, Ira D. "A Short Perspective on Domain Analysis," Univ. of CA at Irvine, position paper for domain analysis working group session at OOPSLA 88.

[BEL79] Belady, L. A. "Evolved Software for the 80's," *IEEE Computer*, February 1979, pp. 79-82.

[BIG88] Biggerstaff, Ted J. "The Nature of Semi-Formal Information in Domain Models," MCC, 1988.

[BOO87] Booch, G. *Software Components with Ada*, Benjamin/Cummings, Menlo Park, CA, 1987.

[BRA85] Christine L., et al. *Ada Reusability Guidelines*, SofTech, In., April 1985.

[COA89] Coad, Peter. "OORA--Object-Oriented Requirements Analysis," manuscript accepted for COMSAC, 1989.

[CTA87a] "An Operational Concept of Software Reuse," Computer Technology Associates, Inc., June 15, 1987.

[CTA88] "Semi-Automatic Development of Payload Operations Control Center Software," Computer Technology Associates, Inc, October 31, 1988.

[DIL85] Dillenhunt, Don, Norman S. Nise and Chuck Griffin. "Reusable Software Development," STARS Application Workshop, April 1985.

[DOD86] Department of Defense (DoD). *Reusability Guidebook*, STARS- 4th Applications Workshop, San Diego, CA, September 15-19, 1986.

[DOD88] *Military Standard Defense System Software Development*, DoD-STD-2167A, Department of Defense, February 29, 1988.

[FRE84] Freeman, Peter. *Analysis of the Draco Approach to Constructing Software Systems*, University of California at Irvine Technical Report 85-07, 1984.

[FRE87] Freeman, Peter. "A Conceptual Analysis of the Draco Approach to Constructing Software Systems," *IEEE Transactions on Software Engineering*, Volume SE-13, Number 7, July 1987.

[GAR88] Gargaro, Anthony. "Analysis of the Impact of the Ada Runtime Environment on Software Reuse," prepared for CECOM/CSE, DAAB07-85-C-K524, September 30, 1988.

[GRE88] Greenspan, Sol J. and Mark Feblowitz. "Conceptual Modeling for Software Design," *Proceedings of the Workshop on Automating Software Design*, St. Paul, MN, August 25, 1988.

[ISC88] Iscoe, Neil. "Domain Models for Program Specification and Generation," position paper for domain analysis working group session at OOPSLA 88.

[JAC87] Jacobson, Ivar. "Object-Oriented Development in an Industrial Environment," *OOPSLA '87 Proceedings*, Association for Computing Machinery, October 4-8, 1987, pp. 183-191.

[JAL89] Jalote, Pankaj. "Functional Refinement and Nested Objects for Object-Oriented Design," *IEEE Transactions on Software Engineering*, Vol. 15, No. 3, March 1989, pp. 264-270.

[LAN79] Lanergan, Robert G., and Brian A. Poynton. "Reusable Code--The Application Development Technique of the Future," *Proceedings of the Joint SHARE/GUIDE/IBM Applications Development Symposium*, October 1979.

[LIE88] Lieberherr, I. and Holland A. Riel. "Position Paper: OOPSLA 1988 Workshop on Domain Analysis," Northeastern Univ., September 5, 1988.

[LON86] London, Philip E. and Martin S. Feather. "Implementing Specification Freedoms," *Reading in Artificial Intelligence and Software Engineering*, Charles Rich and Richard Waters (eds.), Morgan Kaufman Publishers, Los Altos, CA, 1986.

[LUB88] Lubars, Mitchell D. "Domain Analysis and Domain Engineering in IDeA," Microelectronics and Computer Technology Corp., position paper for domain analysis working group session at OOPSLA 88.

[MAR88] Marca, David A., and Clement L. McGowan. *SADT™ Structured Analysis and Design Technique*, McGraw-Hill Book Company, 1988.

[MCC85] McCain, Ron. "A Software Development Methodology for Reusable Components," STARS Application Workshop, April 1985.

[MCF85] MacFarland,Clay and Terry Rawlings. *DARTS: A Software Manufacturing Technology*, General Dynamics Data Systems Div., Western Center, San Diego, CA. Presented at the STARS Application Workshop, April 1985.

[MCG88] McGregor, John D. "Domain Analysis Position Paper," Murray State Univ., presented at domain analysis workshop at OOPSLA '88.

[MCN86a] McNicholl, Daniel G. and Palmer, Constance, et. al. *Common Ada MIssile Packages (CAMP) Volume I: Overview and Commonality Study Results*, McDonnell Douglas Astronautics Company, AFATL-TR-85-93, May 1986.

[MCN86b] McNicholl, Daniel G., Constance Palmer, et. al. *Common Ada MIssile Packages Volume II: Software Parts Composition Study Results*, McDonnell Douglas Astronautics Company, AFATL-TR-85-93, May 1986.

[MOO88] Moore, John M. and Bailin, Sidney C. "Position Paper on Domain Analysis," CTA Inc., prepared for domain analysis working group session at OOPSLA '88.

[NEI81] Neighbors, James M. *Software Construction Using Components*, University of California at Irvine Technical Report 160, 1981.

[NEI83] Neighbors, James M. "The Draco Approach to Constructing Software From Reusable Components," *Proceedings of the Workshop on Reusability in Programming* (sponsored by ITT), September 1983, pp. 167-178.

[NEI88] Neighbors, James. "Report on the Domain Analysis Working Group Session," based on meeting at OOPSLA '88.

[NIS86] Nise, Norman S. and Chuck Giffin. "Considerations for the Design of Ada Reusable Packages," *First International Conference on Ada Programming Language Applications for the NASA Space Station*, Volume II, June 2-5, 1986, pp. E.1.6.1-E.1.6.10.

[PAR88] Parsaye, Kamran, "Acquiring and Verifying Knowledge Automatically", *AI Expert*, May 1988.

[PRI87] Prieto-Diaz, Ruben. "Domain Analysis for Reusability," *Proceedings of the Eleventh Annual International Computer Software and Applications Conference* (COMPSAC 87), Tokyo, Japan, October 7-9, 1987, pp. 23-29.

[PRI88] Prieto-Diaz, Ruben. "In Search of Tools for Domain Analysis," GTE Laboratories, position paper for domain analysis working group session at OOPSLA '88.

[PYS88] Pyster, Arthur and Barnes, Bruce. "The Software Productivity Consortium Reuse Program," *Digest of Papers COMPCON 88*, IEEE Computer Society, pp. 242-248.

[QUA88] Quanrud, Richard B. *Generic Architecture Study*, Report 3451-4-14/2, SofTech, Inc., January 22, 1988.

[ROB88] Robinson, William N. "A Use of Domain Goals in Specification Construction," Univ. of Oregon, position paper for domain analysis working group session at OOPSLA 88.

[RTE87] Materials from the Reuse Tools and Environments Working Group meeting, held at the Tenth Minnowbrook Workshop on Software Reuse, July 28-31 1987.

[RTE88] Materials from the Reuse Tools and Environments Working Group meeting, held in Bass Harbor, Maine, June 22-24, 1988.

[SAY88] Sayrs, B. and X. Peltier. "Taxonomic vs. Non-Taxonomic Datatypes," Schlumberger Austin Systems Center, position paper for domain analysis working group session at OOPSLA 88.

[SEI88a] "Reusable Software Development Process," Software Engineering Institute, Carnegie Mellon Univ., presentation to Affiliates Workshop, 1988.

[SEI88b] "Software Engineering Reuse Experiments, Software Engineering Institute, Carnegie Mellon Univ., presentation to Affiliates Workshop, 1988.

[SIM88] Simos, Mark. "Workshop on Reuse and Tools Environments: Approach and Goals," Unisys Defense Systems, position paper for *Domain Analysis* Working Group session at OOPSLA 88.

[SMI88] Smith, Mark K. "An Integrated Approach to Software Requirements Definition Using Objects" *Tenth Structured Development Forum (SDF-X)*, San Francisco, CA, August 8-11, 1988.

[SPS87] Grau, J. K., E. Comer and G. Ivie. *Knowledge-Based Reusable Software Synthesis System*, NASA Contractor Report 178398, Software Productivity Solutions, Indialantic, FL, 1987.

[SPS88] Classic-Ada[tm] User Manual, Software Productivity Solutions, Indialantic, FL, 1988.

[STA85] STARS Application Area Workshop, Working Group Reports, April 1985.

[UNI88] "Reusability Library Framework," Unisys Defense Systems, position paper submitted for Domain Analysis Working Group session at OOPSLA 88.

[WOO88] Woodfield, Scott N. and David W. Embley. "Modeling Reality," Brigham Young Univ., position paper for domain analysis workshop at OOPSLA '88 conference.

Appendix C

Synopses of Domain Analysis Research

This appendix contains detailed synopses of written materials about domain analysis. The synopses are grouped according to the organization performing research and development about domain analysis. When this report was written several organizations had published the following information about their efforts in domain analysis:

- o Brigham Young University - Domain modeling.
- o Computer Sciences Corporation - Impact of the Ada runtime environment on parts in a domain.
- o Computer Technology Associates - Domain analysis case history for NASA mission ground control applications, use of object-oriented methodology, Software Reuse Environment.
- o General Dynamics - Domain modeling, archetypes, DARTS software generation system.
- o GTE Laboratories - Domain analysis process, component classification, tools for domain analysis, conceptual modeling platforms.
- o IBM - Domain analysis process, domain analysis study for NASA space shuttle
- o Knowledge System Corporation - Domain modeling.
- o McDonnell Douglas Astronautics Company - domain analysis process, domain modeling, commonality analysis, component development, Automated Missile Packages Engineering Expert, domain analysis case history for Air Force missile systems (CAMP program).
- o Microelectronics and Computer Technology Corp. - Domain modeling, DESIRE design recovery system, domain analysis process, IDeA design environment.
- o Murray State University - object-oriented modeling and specification techniques for domain analysis
- o Northeastern University - Domain modeling, Demeter object-oriented programming system.
- o OOPSLA Domain Analysis Working Group - Domain analysis process, a small domain analysis experiment.
- o Reuse Tools and Environments Working Group - Domain analysis process, domain analysis tools.
- o Rockwell International - Reuse process, commonality analysis.

- o Schlumberger Austin Systems Center and Schlumberger-Doll Research - Domain knowledge acquisition, domain-specific automatic programming, UNIX project, taxonomic models for domain classification.

- o Software Engineering Institute - Domain analysis process, reanalysis of CAMP effort.

- o SofTech - Generic architectures, Ada reusability guidelines, domain analysis process.

- o Software Productivity Consortium - Domain analysis process, module families, canonical designs.

- o Software Technology for Adaptable, Reliable Systems Program - Impact of domain on parts development and reuse libraries.

- o Special Interest Group for the Ada Programming Language, Reuse Working Group - Domain analysis and reuse libraries.

- o Unisys Defense Systems - Reusability Library Framework, domain modeling, domain analysis process.

- o University of California at Irvine - Domain modeling, domain analysis process, domain knowledge acquisition, Draco methodology and toolset.

- o University of Oregon - Domain modeling.

The reader should note that many reports discuss technology areas related to domain analysis (such as object-oriented methodologies, reuse libraries and program generation systems), and this discussion is often included in the synopses. While research into these related areas is important, it was not the focus of the survey. As such, the reader may find the synopses represent an incomplete or imbalanced perspective in many areas.

C.1 Brigham Young University

Woodfield and Embley [WOO88] reinforce the importance and difficulty of domain analysis in the context of software development and state "The standard approach to software development has been:

1. analyze and understand the real-world problem,
2. abstract the problem and convert its essence to software,
3. execute the software program created,
4. translate the result back to reality."

They identify the software developer's inability to abstract reality as the core problem, and propose that this is "largely a result of using an inadequate domain model." They define a domain model to be "inadequate when there are important aspects of reality that cannot be represented." As an example, they cite the concurrent nature of many real-world problems contrasted with the sequential nature of the domain model used by most developers.

They note that domain models are not explicitly incorporated into the development paradigm, but are implicitly used nevertheless. They believe the domain model has an impact whether it is explicit or implicit, and the effect of an inadequate model is the same in either case.implicit.

The authors document concern that even developers who consciously use domain models do not recognize the lack of expressive power afforded by current development paradigms. The developers record only the information that can be used by these paradigms, ignoring concepts that are not easily represented in some design or programming language. They cite the use of fuzzy logic as an example of a model often automatically filtered out when developing the abstractions.

Woodfield and Embley conclude that "[W]hat is needed are better domain models, better representations for domain models, and better methodologies for developing good domain models. Good domain models and their corresponding representation must be created first. No methodology, no matter how good, will be able to make up for an inadequate domain model."

C.2 Computer Sciences Corporation

"Analysis of the Impact of the Ada Runtime Environment on Software Reuse" makes several points relevant to performing domain analyses for systems to be implemented in the Ada programming language.

"[T]he inability to adequately express time-critical requirements presents a problem that pervades many of the issues for software reuse from domain analysis to the choice of design and implementation techniques. For example, if a part is to be reused in deadline driven applications, the execution time of the part is likely to be an important attribute that must be accurately specified with respect to the requirements of the application domain that are included in the documentation for a part. Unless a part is adequately documented its potential for reuse is severely compromised. The documentation, at a minimum, must specify the function performed by the part and how the part may be reused within an application domain."

"The capacity transparency of the model to the target execution environment is problematic. The development of reusable parts without regard to whether the target execution environment is a single processor or a configuration of loosely coupled computers does not appear to be practical. Once the possibility of distributing the execution of an Ada program is raised, the issue of program partitioning becomes significant. Partitioning strategies may be attributed as either 'post-partitioning' or 'pre-partitioning'. Each strategy requires RTE support that may affect software reuse when, for example, objects are located such that referencing them incurs an unacceptable performance efficiency overhead. One rudimentary scheme proposed for developing reusable parts for distributed execution environments using pre-partitioning requires adherence to specific guidelines for composing reusable applications. However, the scheme assumes the cooperation of the RTE to support remote references across loosely coupled computers."

"The abstraction of the traditional functions performed by specialized executives into Ada constructs serviced by the RTE would seem to guarantee the desired predictability and reliability of a part over its application domain. Unfortunately, some Ada constructs, particularly those having implied temporal semantics, are specified at either an inappropriate level of abstractions resulting in restricted functionality or permissive semantic specifications. This compromises uniformity of implementation among RTEs and may lead to software that is less reusable as a result of explicit or implicit dependencies upon the RTE that are not specified by the application domain."

"These dependencies may result in timing behavior of a part that may be neither predictable nor reliable outside of the application in which is was originally developed. This condition is further exacerbated by the lack of formality in the requirements specification of applications domains for embedded real-time MCCR systems. The documentation for a part will most likely propagate this informality in the most critical area of timing behavior, thereby camouflaging its lack of predictability and reliability."

Gargaro also notes that subtle dependencies on the RTE can cause problems "when combining reusable parts in an execution environment that is different from those used to originally develop the parts." As an example, two parts from different applications are to be combined in a new application in a new environment. Each application individually may have been ported to the new environment before, so it is known that the parts can be moved. However, this may install a false degree of confidence "that the parts are free of RTE dependencies. It is only when combining those parts that are dependent upon conflicting RTE behavior will failure or aberrant execution occur."

A related effort by Gargaro addresses the impacts of real-time systems on software reuse, with the product being an annex to his "Ada Reusability Handbook". Four areas were identified as critical to reuse for Ada real-time systems:

1. Distributed Execution - a partitioning strategy based upon "virtual nodes" was recommended

2. Scheduling - provides advice for developing "reusable paradigms"

3. Error Recovery -[still to be done]

4. Resource Control - provides advice for developing "reusable paradigms"

Reusable paradigms are templates for Ada interfaces providing particular capabilities required by real-time systems. Gargaro has found that "formulating concise standalone guidelines for developing reusable real-time software parts is difficult" and concluded that "developing templates for proven real-time paradigms may be more practical."

C.3 Computer Technology Associates

Bailin and Moore are currently working on an automated reuse system for Computer Technology Associates. In [BAI88], Bailin and John Moore describe a process model for software reuse, where the activities are divided up into two sides: a *supply side* and a *consumer side*. These are the supply side activities:

- o **Analyze** domain
- o **Develop/refine** reusable products
- o **Classify** and **store** reusable products

On the consumer side we find these activities:

- o **Specify** requirements for a product
- o **Search** for candidate reusable products
- o **Evaluate** candidate products
- o **Integrate** reusable products with new software

Tools to help the user create entity-relationship (ER) diagrams, data flow diagrams, and object diagrams are integrated with the SRE.

Searches for reusable components can be initiated from any of the diagrammatic elements *representing the system under construction*. The user can then display information about a product, select one or more of the products (the tool does not extract the product, but adds a note that the product is to be reused on the new system), browse through related products, or browse through related keywords (perhaps finding new keywords on which to initiate a search).

Moore presents the objectives, partial results and current efforts of CTA in supporting NASA GSFC with software development methods and environments. [MOO88]The methods developed by CTA include the GOOD (Goddard Object-Oriented Design) method and OORA (Object-Oriented Requirements Analysis) method, both of which are supported by the SRE (Software Reuse Environment). "The most recent phase of our work has been a Domain Analysis of Space Mission Ground Control Center software. Our goal is to populate the Reuse Database, which is the core of the SRE, with descriptions of the items identified through the Domain Analysis." Of the four systems examined to date, two systems were "developed to serve as repositories of reusable software for Control Center software" (the other two systems make use of those components). As such, a domain analysis has already been "implicitly performed in the specification ... and we decided to build on the knowledge embodied in those systems." The following material summarizes the experiences and lessons learned about domain analysis.

The Domain Analysis Process

"Domain Analysis is the process of formulating the commin elements and structure of a domain of applications. The purpose of a domain analysis is to provide a *framework for systematic reuse* of software products, as well as a *repository of reusable products* ... Our methodology for developing domain models can be summarized in the following steps:

1. Construct a strawman generic model of the domain by consulting with domain experts.

2. Examine existing systems within the domain and represent this understanding in a common format (e.g., E-R diagrams) and textual descriptions. (Part of this examination should be to identify objects within the domain.)

3. Identify similarities and differences between the systems by consulting with domain experts. This analysis should identify keywords that express these relationships.

4. Refine the generic model to accommodate existing systems.

5. Populate the Reuse Database with component and system descriptions."

Moore and Bailin note the similarities between these steps and those for knowledge acquisition for expert systems and suggest that some tools and techniques be adapted for knowledge acquisition to support domain analysis.

Domain Classifications and Boundaries

Because their domain was defined by their customer (NASA), they "have not directly addressed the issue of how a domain is identified ... We take the pragmatic view that a *domain is any problem space encountered by a developer*. Frequently encountered, large problem spaces (e.g., Control Centers within NASA) seem to be good candidates for domain anlysis. This is based on the belief that the overhead incurred during the analysis of such systems can be compensated by productivity increases in future developments."

They "did encounter issues in setting the boundaries of the ... domain." They cited an example of a facility that is operationally an essential part of the domain, but which also "has its own set of typical data structure and algorithms and is typically developed by a dedicated ... organization." As such, they had valid reasons both for including and excluding the facility within the domain. They "resolved this question pragmatically; [they] were already confronted with a massive amount of information to be digested, correlated, and abstracted from, and [they] decided not to further complicate the field." As a result, they chose to regard the facility as a "distinct (although related) domain, not subsumed by the ... domain."

Domain Models

"The result of the domain analysis is the domain model. Within our work, this model consists of *components, relationships, attributes and keywords*. Components may be subsystems, entities, functions, or data structures. Relationships specify how systems, subsystems, and components are structured. Attributes provide descriptive information about the systems, subsystems, and components. Keywords are used to classify components from the perspectives of the various users (i.e., developer, tester, end-user, itc). Classification is especially important for software reuse because it permits the developer to choose between similar but subtly different products."

They provide descriptions of this information in "two forms: entries in the Reuse Database and Entity-Relationship Diagrams (ERDs). The Reuse Database serves as an interactive version of the domain model, allowing developers to identify and locate required items dynamically ... The Reuse Database supports multiple views of the same system, subsystem, or component. A given subsystem, for example, may have a requirements specification, design, code, and/or test view.

Each view captures those aspects of the subsystem that pertain to a given development phase. By linking all views to the same element in the Domain Model, it is possible to identify elements for reuse early in the development process and then reuse later views as well."

"The E-R diagrams serve as a more tangible complement to the Reuse Database. They provide a means of recording information in the domain model prior to populating the Reuse Database. They also provide an effective means of communicating the model's contents to domain experts (during the domain analysis) and to software developers (who use the domain model). The E-R diagrams provide a *uniform representation* of the domain. This is particularly important when individual systems are specified in different languages or according to different paradigms. The uniform representation facilitates comparisons between systems and between their components."

Objects (What they are and how we find them)

An object is any data item (or lower-level object) together with the functions that act on the data, such that:

1. The functions are *tightly coupled* to the data.

2. The functions *may* be tightly coupled to each other.

3. The functions are *not* tightly coupled to anything else."

The authors identify a few heuristics for identifying tight coupling. "[W]e have found that using [these heuristics] we can decide in a straightforward manner how to group functions and data [that is, to identify appropriate objects]."

"In our domain analysis ... we examined existing requirement specifications which were in the form of Data Flow Diagrams and Functional Block diagrams. Although these diagrams were not specifically object-oriented, for the most part we were able to infer objects from them."

In some cases, they found it necessary to create new entities by grouping certain functions and data together. They "did not address the question of whether the objects we identified were *good* abstractions. We limited ourselves to imposing structure on the information already present in the system specifications." Making the abstractions 'better' is to be addressed when "these systems are fed back into the generic model (Step Four of our analysis process, which has not yet been performed)."

The authors feel that it is at this last step "that the real difficulties in defining objects will arise. Our own development experiences suggest that the most tricky aspect of defining objects is how rich to make them. The question is whether to implement a 'spanning set' or a 'rich set' of functions and data for an object. A 'spanning set' is a minimal set of functions and data, which can be combined to achieve all required operations on (or by) the object. A 'rich set' contains a spanning set plus additional operations and/or data that are typically needed by users of the object. The heuristics for tight coupling ... may suggest a layering of object abstractions so that both richness and orthogonality are achieved. We have found, however, that this is an area requiring a considerable amount of trial and error as well as engineering judgment."

"Semi-Automatic Development of Payload Operations Control Center Software" [CTA88]summarizes Computer Technology Associates' investigating methods and tools for automating software development at Goddard Space Flight Center. The report addresses the most recent phase of this effort, which was a domain analysis of the payload operations control center (POCC) software. It also proposes an approach to semi-automatic development of POCC Application Processor (AP) software.

"The domain analysis enabled us to abstract from specific systems, the typical components of a POCC AP. We were also able to identify patterns in the way one AP might be different from another. These two perspectives -- aspects that tend to change from AP to AP, and aspects that tend to remain the same -- suggest an overall approach to the reuse of POCC AP software."

"We found that the different parts of an AP require different development technologies. We propose a hybrid approach that combines *constructive* and *generative* technologies. Constructive methods emphasize the assembly of pre-defined reusable components. Generative methods provide for automated generation of software from specifications in a very-high-level language (VHLL)." Techniques used in automating support for constructive approaches include:

- *Dialog-based specification*: "Automated assembly of components, based on question-and-answer interaction with the developer. Suitable for configuring the top level of an AP." Rules for integratin subsystems could also be deduced from relaitonships present in the Reuse Database.

- *Interactive selection of components*: "Navigation of available components in the Reuse Database. Suitable for configuring standard interfaces (such as Operator input/Output) and for selecting standard processing functions (such as Telemetry processing)."

Techniques used in automating support for generative approaches include:

- *Domain-specific Very-High-Level Language* (VHLL): "Suitable for generating the AP database together with those components that access the database." CTA recommends the "pre-spec" method, where Ada itself is used to express high level processing, rather than defining a new unique VHLL for each application domain.

- *Graphical programming*: "Automated source code generation from diagrams. Suitable for assembling new combinations of interactively selected components ... Object Diagrams would be used to combine or integrate reusable packages. Dataflow Diagrams would be used to integrate reusable functions and procedures."

They scoped the domain analysis to seven operations control systems, two of which are baselines used in building POCC APs. "The APs for specific missions are built by adding to, modifying, and/or replacing parts of the baseline software." The rest of the systems are based upon the baselines. This permitted an analysis of existing reusable software, and its use or non-use in subsequently developed application systems.

The analysis of existing systems was limited to the requirements specifications for those systems for several reasons: limitaitons of available resources to perform the domain analysis, an understanding of system requirements is necessary to understanding a domain, and "because similar requirements may be realized by apparently dissimilar software designs and implimentations."

The result was a focus on high-level reusable components, which the authors predict will provide the greatest leverage in terms of reuse payoff. They advocate the use of object-oriented design (OOD) techniques for the new reusable baseline because "OOD facilitates reuse of high-level components by establishing unambiguous interfaces and control relationships."

The existing systems which were studied were similar in their capabilities, but they concluded that reusing any individual system would be difficult because thy did not have ofject-oriented interfaces.

Each of the existing specifications was mapped to an object-oriented interface to produce a set of specifications with a common underlying model.

An analysis of the systems yielded the following six types of variations among systems in their domain:

- o External interfaces
- o Devices supported
- o Data definitions
- o Processing performed
- o Placement of functions
- o Leveling

While they found a "fairly constant" allocation of functions to AP subsystems, the exact partitioning into subsystems did vary. The components are found at a level immediately below subsystems. Important characteristics of components included whether they were *optional, stable,* none, or both. Optional components are components which may or may not be present in a typical application system. Stable components are those which require little, if any, modification to support a new mission. Another category of components was identified where a *range* of stable options could possibly be needed. They found the classificaiton of internal functions and structures of some systems to be difficult. This was because the granularity and ordering of the functions, and details of how they were implemented differed across applicaitons. They recommended the parameterization of components across missions, and standardization of low-level functions as possible solutions. Tools that facilitate the composition of low-level reusable functions into mission-specific higher-level functions were also recommended.

The authors then identified three major technical obstacles to software reuse as evidenced by the systems thy analyzed: mission-specific requirements, obsolescence of host systems and external interfaces, and increasing complexity of system requirements.

Concentration on the subsystems most likely to be reusable, layering of subsystems, and tailorability were all identified as techniques for dealing with mission-specific requirements, but the recommended solution was the use of automated source code generation techniques. An example is the automatic generation of source code to access mission-specific databases.

One of the standard baselines became obsolete due to a change in the required host system and other external interfaces. The second baseline is expected to become obsolete due to requirements for interfacing with new devices, networks, and external systems that were not present in earlier missions. The second baseline is also tied to obsolete technologies in the areas of user interfaces (text-based versus graphical input/output), hardware technology, and distribution of control among user site. They recommend increased standardization and development ofvirtual interfaces to remedy the potential problems. The use of object-oriented desin is also recommended at a strategy for deriving more maintainable designs.

The functional requirements for new systems are becoming more and more demanding, leading to the development of increasingly complex software. "Evolving complexity poses an especially difficult problem for reuse because it is impossible to predict the manner in which complexity will evolve. Rather than build into the development environment an expected evolution of POCC APs, which is almost guaranteed to be wrong, we propose to provide meechanisms for *handling* evolving complexity." Layered architectures is one method of dealing with complixity

becoming more popular. The authors propose the use of graphical programming solution. They also describe the applicaiton of a *pipeline processing paradigm* for handling evolving complixity of telemetry subsystems. The authors reference a system they developed for STARS which can automatically generate code from a pipeline description of such a subsystem.

Figure C.3-1 illustrates the integrated concept for semi-automatic development of POCC AP Software. However, the "goal is not so much to automated as to raise the level of abstraction at which developers must think."

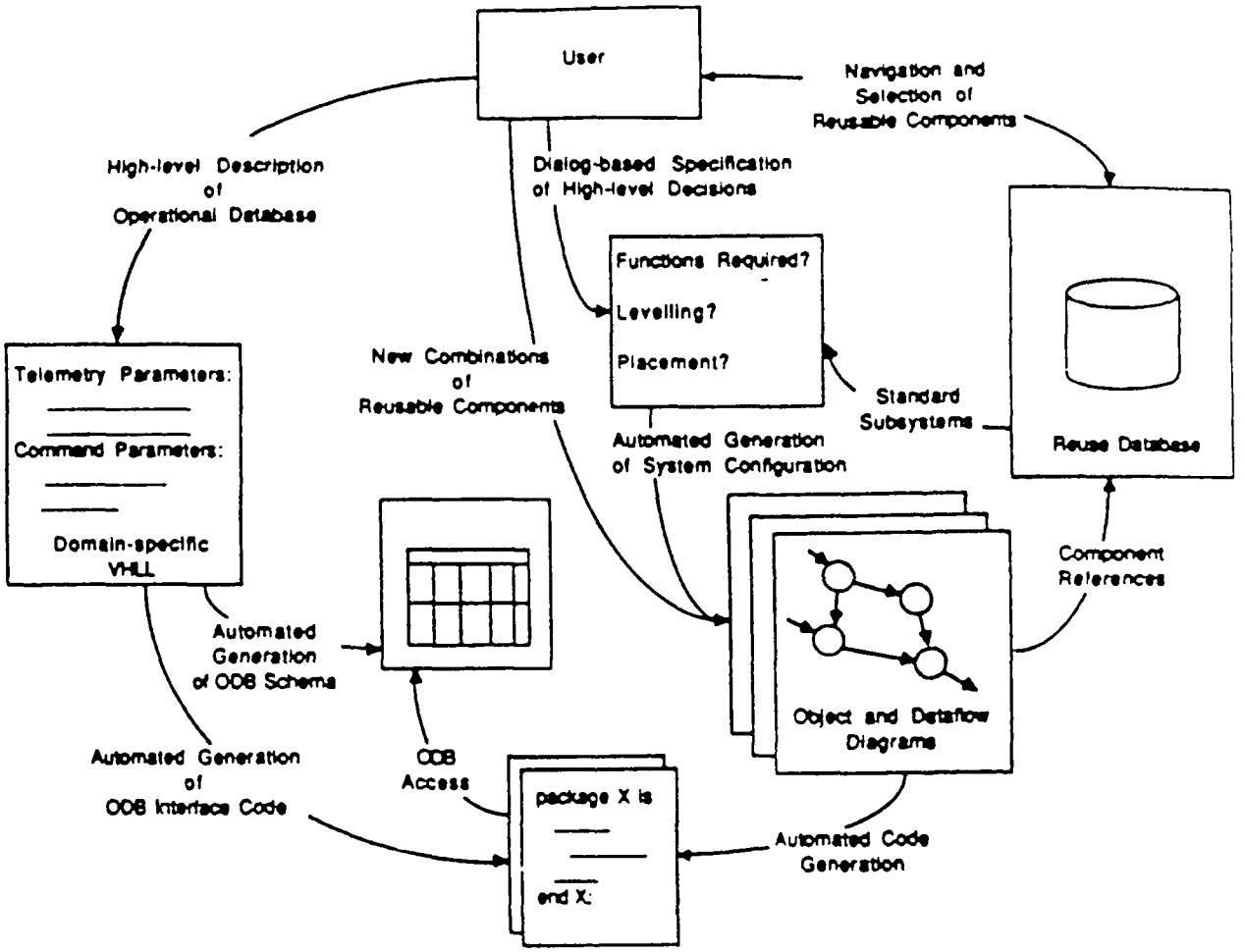


Figure C.3-1. Combination of Four Automation Techniques

C.4 General Dynamics

McFarland and Rawlings describe an approach for the use of DARTS (Development Arts for Real-Time Systems) which could provide for the generation of alternative programs within the context of a given application domain. [MCF85]

The first time a new system is to be built, an analyst would perform a domain analysis and then design a general solution to the problem, called an *archetype*. Alternatively, if a working system already exists in this problem class, then this system can be modified to be an archetype. Once an archetype exists, new systems would be developed using the original archetype as a prototype.

A domain-specific language and translator must be developed as part of the process of domain analysis and solution design. Knowledge bases representing the application domain must also be developed. The result of the domain analysis phase would be the generation of a new software development environment tuned to that application domain.

Based on interviews with developers and users of this system, it should be noted that actual usages of the system have been bottom-up, with no application of domain analysis. We are further investigating the use of DARTS technology in addressing the differences between similar applications.

C.5 GTE Laboratories

In 1987, Ruben Prieto-Diaz investigated some of the current efforts in domain analysis and concluded that "[d]domain analysis is a knowledge intensive activity for which no methodology or any kind of formalization is yet available. Domain analysis is conducted informally and all reported experiences concentrate on the outcome, not on the process." [PRI88]

Prieto-Diaz informally defines domain analysis (DA) as "an activity occurring prior to systems analysis and whose output (i.e., a domain model) supports systems analysis in the same way that systems analysis output (i.e., requirements analysis and specifications document) supports the systems designer's tasks." He finds the two activities are similar because they both involve developing a model for an application system. He finds the two activities are different because with domain analysis one tries to generalize all the systems in an application domain rather than focusing on a specific system. "Domain analysis is thus at a higher level of abstraction than systems analysis."

The process of domain analysis results in the identification of objects and operations common to all systems within the same domain and the definition of a model to describe their relationships. The model is defined using a domain specific language "with syntax and semantics designed to represent all valid actions and objects in a particular domain." New applications systems in that domain can then be defined using the language. "This is the *reuse of analysis information*, and in our opinion is the most powerful sort of reuse."

Key domain analysis issues briefly mentioned in the paper include: the definition of domain boundaries, interconnecting domains "so that modeling a system in one domain translates to objects and operations in another domain," dealing with objects at different levels of abstraction, and finding all possible operations on objects and relationships between objects.

According to Prieto-Diaz, the "crux of the problem" in domain analysis is analyzing and defining these domain languages, an activity requiring the use of experts. One of the roles of the experts is to provide knowledge about the domain. "There is a strong relationship between DA and knowledge acquisition. Building a knowledge base and defining heuristics for an expert system are basically the same problems as DA."

The author gives considerable attention to the issue of classification and its relationship to the domain analysis process. "Classification is the product of integrating the knowledge inherent to the objects of the collection with the knowledge of the domain where the collection is going to be used. With classification we are identifying the objects and operations of a domain. We are also identifying the common characteristics of these objects and operations, which allows us to make generalizations about them."

He compares the process of domain analysis to the process of developing faceted schemes in library science (called *literary warrant*), and concludes that the activities performed during the process of literary warrant "are basically the same activities conducted in domain analysis for reusability." As an example, he shows how building a classification scheme can provide a primitive domain language.

Two case studies in software reusability and two existing proposals for domain analysis were investigated by Prieto-Diaz. These include the Raytheon and CAMP experiences, and the research performed by McCain and Arango. His findings are summarized in the next paragraph.

The first case study discussed in the paper was the Raytheon experience, where a methodology was developed to reuse code in the business applications domain. "We can trace their DA process through the following activities:

o Identify common functions across business applications

o Group them by classes

o Organize into a library

o Analyze business systems structures

o Identify common structures

o Abstract such structures

o Use structures and components to build new systems"

Two interesting discoveries made by Raytheon were that "most modules fall into only three major classes," and "most programs fall in one of three 'logic structures' or a combination of them (they have defined 85 different logic structures combinations)."

"NEC of Japan has followed a very similar approach, albeit independent from Raytheon Their DA process can also be traced through the same activities we listed for the Raytheon experience."

Prieto-Diaz characterizes the approach used by the CAMP developers "by the following activities:

o Identification of similar systems

o Functional decomposition

o Functional abstraction

o Interface definition

o Component encapsulation"

His evaluation of McCain's reuse studies identifies domain analysis as a key task in the reuse paradigm. He determined that domain analysis is divided into three activities:

o Define domain objects and non-objects (collections of related independent operations)

o Define common operations on objects and object to object relationships (called reusable abstractions)

o Classify the abstractions, resulting in a hierarchy "that supposedly is used as a navigation aid in the selection of reusable components"

"After completing DA, McCain proposes to do component DA which is basically a DA of each identified abstraction to define an abstract interface, essential constraints, limitations, and customization requirements. This process ensures proper encapsulation and is intended to guarantee reusability."

Prieto-Diaz seems particularly critical of McCain's failure to describe how to perform the various activities, and McCain's failure to clearly describe the outputs of each activity.

He feels that "Arango proposes a more systematic approach to domain analysis" with well-defined activities and workproducts, supporting "a controlled transition from initial analysis to the actual encapsulation of reusable components." Here is Prieto-Diaz's summary of Arango's steps:

- o Define the domain boundaries
- o Collect standard examples of implementations and perform a systems analysis of each
- o Identify abstractions and express them formally using conceptual modeling languages
- o Refine the abstractions to include operational information
- o Collect and encapsulate the abstractions into reusable components

The author notes that both McCain's and Arango's approaches "are in the proposal stage with no examples or test cases provided ... however, we integrated some of these ideas with the domain analysis activities identified in the previous case studies." He refers to the former as "synthetic" DA approaches and the latter as "evidential" DA approaches.

The process model for domain analysis developed by Prieto-Diaz is illustrated in Figures C.5-1 through C.5-5 (he chose DeMarco data flow diagrams to represent the process).

Figure C.5-1 shows the context for the process of domain analysis. "The domain analyst has the procedural know-how on DA. A set of guidelines are used to extract relevant knowledge from the domain expert and from existing systems. The product of DA is a collection of domain specific reusable components and domain standards used by software engineers for building new systems in that domain."

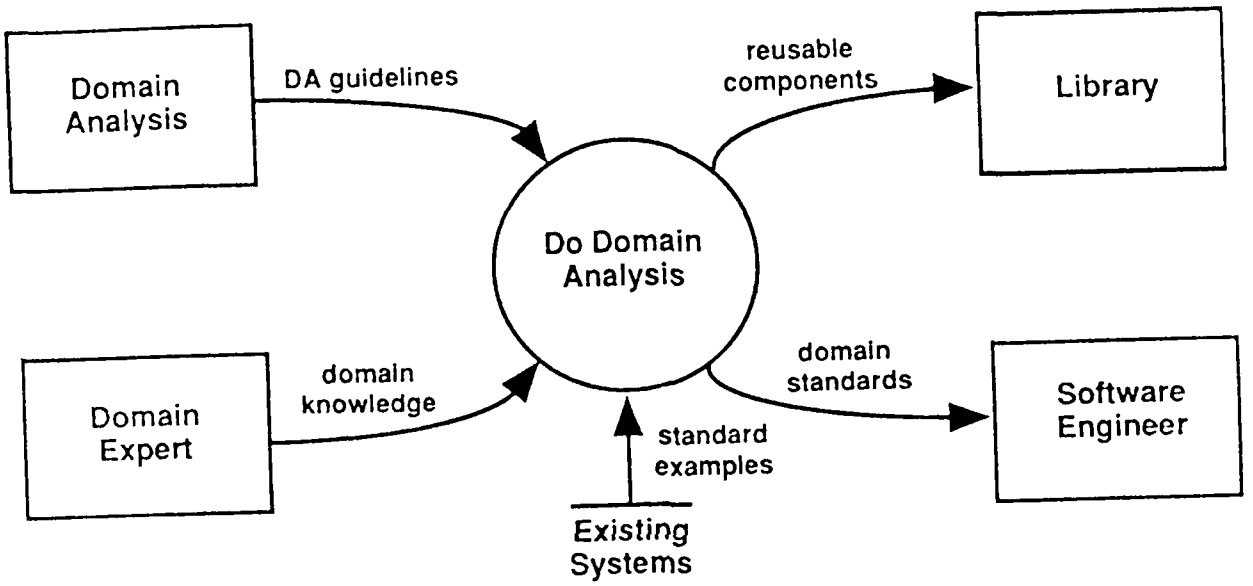
He divides the domain analysis process into *pre-DA activities*, *DA activities*, and *post-DA activities*. The data flows to and from these three high level activities are illustrated in Figure C.5-2.

"Pre-DA activities include defining and scoping the domain, identifying sources of knowledge and information about the domain, and defining the approach to DA." "The DA requirements document is the product of applying general DA guidelines to a well defined and bounded domain." A more detailed representation of the pre-DA activities is provided by Figure C.5-3 (Prepare Domain Information).

The intermediate products produced by the DA activities are the result of three kinds of activities: selection, abstraction and classification.

"The first step in DA activities is to find abstractions [of objects] ... The output of this activity are frames that describe abstractions and that group specific instances of these abstractions. Frames seem to be a natural representation of domain objects and functions and provide the necessary mechanisms to express relationships."

"Classification is the next step in DA. A classification structure can be derived for the complete domain or it can be broken into partial classifications ... The output of this process is a taxonomy of the domain and a structure of relationships that can be used as a model to describe of talk about objects and operations in the domain." "A domain model and a domain language are optional products because not many domains can be modeled and because deriving a domain language is yet a difficult and uncertain activity."



Context Diagram for DA

Figure C.5-1. Context Diagram for Domain Analysis

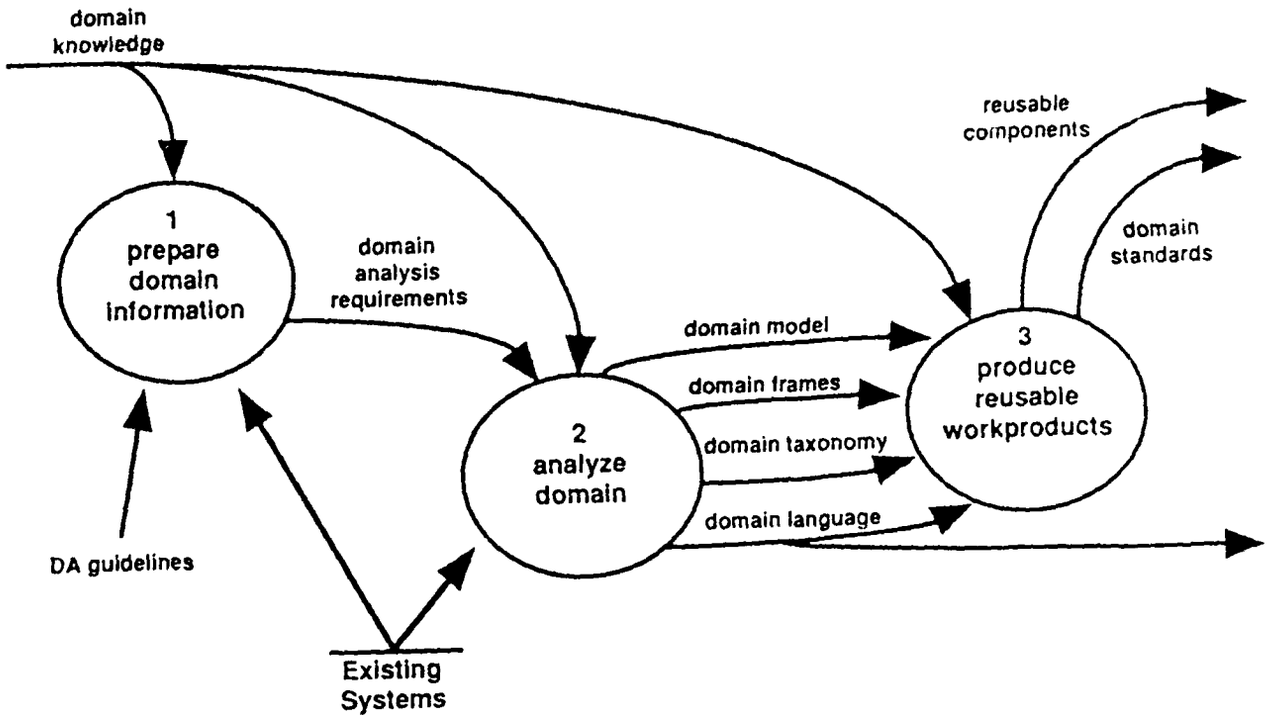


Figure C.5-2. Do Domain Analysis

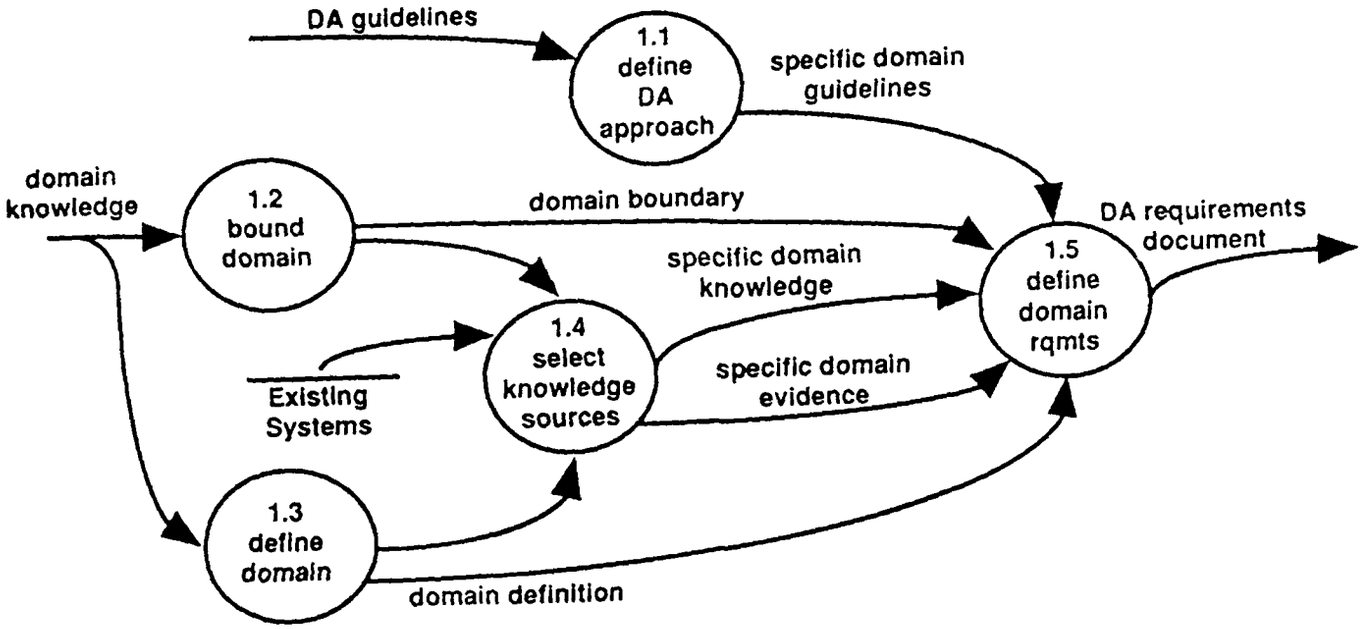


Figure 3.5-3. Prepare Domain Information

Figure C.5-4 (Analyze Domain) shows the specific activities and products involved in domain abstraction and classification.

To identify common features "the domain analyst selects activities common to the existing systems ... This identification process is coupled with the bounding and defining of the domain" (that is, in parallel with certain pre-DA activities).

Abstract objects, functions and relationships are defined in terms of more specific objects, functions and relationships derived from the existing system specifications.

"In classification we create a domain taxonomy from the common objects, functions and their relationships." "A domain taxonomy can have different levels of complexity and can be developed incrementally as domain knowledge is acquired," starting with the most obvious and simple domain classes. Prieto-Diaz proposes that the taxonomy could be a single hierarchy, semantic net, or faceted scheme; it could also be some combination of the three structures. "Selection of a particular representation structure would depend on the kind of domain analyzed. Different forms could be used within the same domain depending on its size. A high level domain model could be in the form of a faceted scheme or a simple hierarchy with semantic nets and frames used for lower level domain elements."

"Post-DA activities include encapsulation and producing reusability guidelines. Encapsulation is the identification and implementation of reusable components based on the domain structure or model ... Reusability guidelines consist basically of documentation for the reuser." Figure C.5-5 (Produce Reusable Workproducts) illustrates the activities and products of the post-DA process.

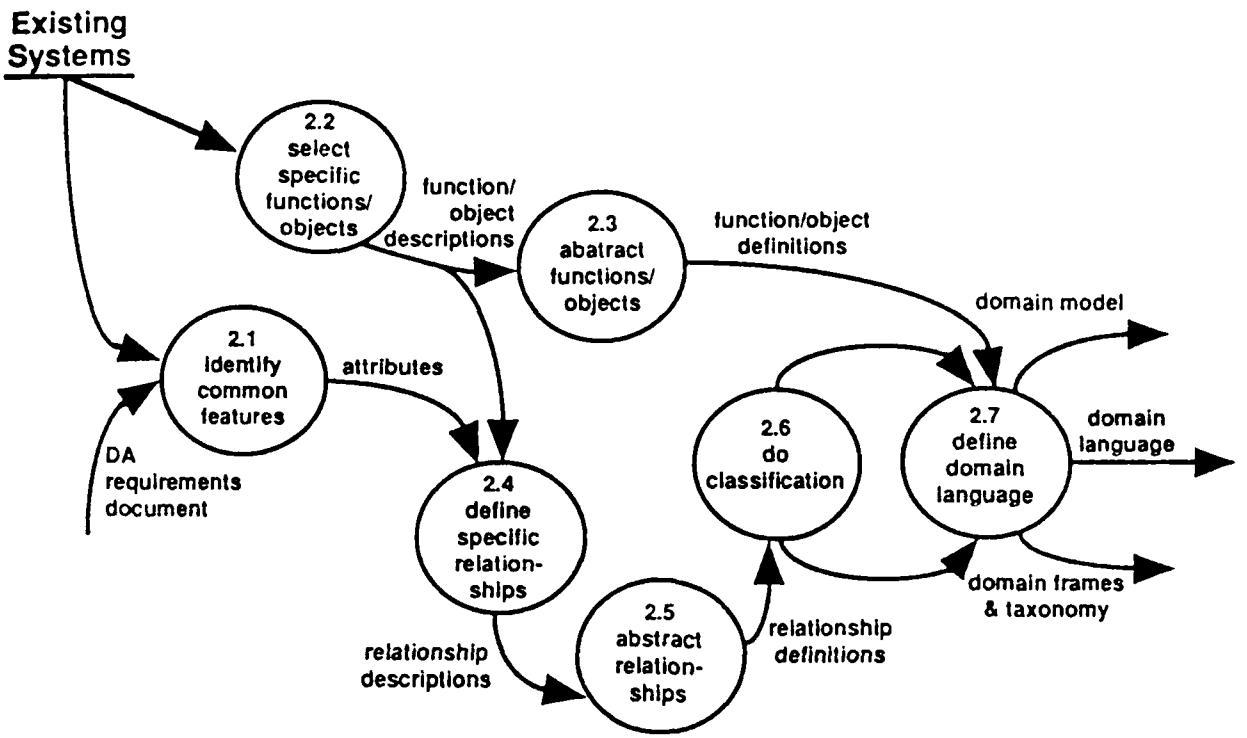
Prieto-Diaz' approach to encapsulation "is similar to that used in object-oriented programming or in McCain's object domain analysis." "Elements that offer the most potential for reusability are selected first for encapsulation." Reusability is achieved through "proper structuring, modularization, and standardization."

"Standards include the set of reusable components available, definitions of logic structures and guidelines to integrate reusable components into systems. Interface and interconnection problems are also part of the domain standards. If a domain language is available, sets of refinements can also be defined" (these would be transformations of the domain language into intermediate and eventually target language representations).

Although not shown by the diagrams, "[i]mplicit in the DA process is a continuous iteration in the domain knowledge acquisition activity in particular and in each of the other activities through feedback from subsequent activities."

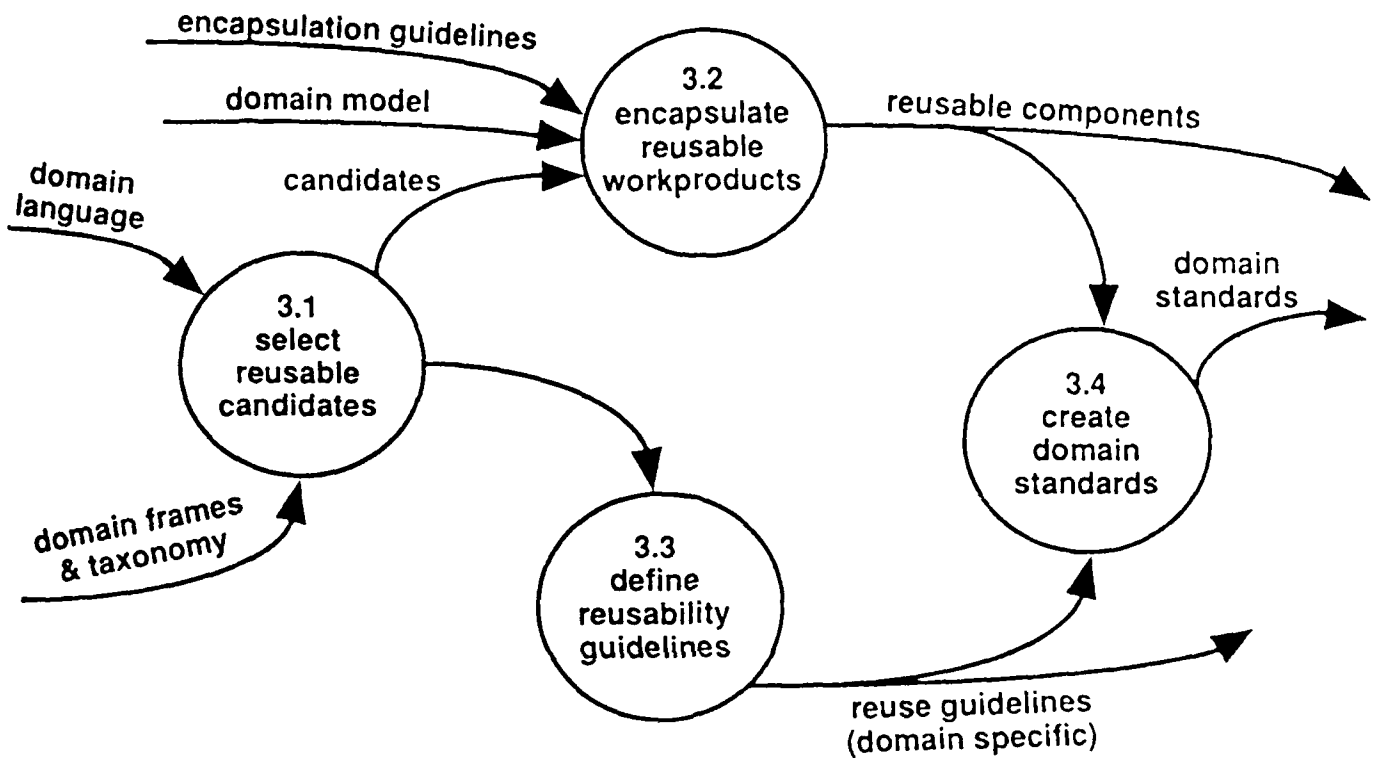
At the time the paper was written, the author was involved in a project at GTE Laboratories to validate the DA process proposed by Prieto-Diaz. They were considering using a telecommunications application as an experiment, and expected to tune the DA process and produce a DA guidelines document. To "monitor the reusability of the resulting components," they planned to use a library system developed by the author. Plans for additional support tools are also an expected result of this effort.

Prieto-Diaz's took the following position for the domain analysis (DA) workshop: "There is a need for tools to support Domain Analysis. We propose the use of existing tools for browsing and manipulating knowledge structures, and the development of tools for conceptual clustering." They are currently "conducting a DA experiment by analyzing a small, well-understood, and well-documented domain of screen editors using EMACS as a reference system." As a result, they have "learned some DA activities can be enhanced substantially with appropriate tools."[PRI88]



Analyze Domain

Figure C.5-4. Analyze Domain



Produce Reusable Workproducts

Figure C-5.5. Produce Reusable Workproducts

Prieto-Diaz notes that it "is difficult to specify tools for a process we do not understand fully. Our approach to specifying tools for DA consists of two steps. First identify some of the primitive activities of the process from the general definition of DA. Then, try to find some tools that support them."

"The DA process identifies common objects, operation, and relationships in a specific applications domain. This process synthesizes domain models that can be represented in different forms and degrees of complexity. Domain models include: domain taxonomies, domain frames, functional models, and domain languages. These models increase reuse leverage substantially by supporting software reusability at higher levels of abstraction."

They learned that "one basic activity, the process of conceptual clustering, repeats many times during the process of discovering common objects and operations. This activity is a grouping, and regrouping, of objects and operations by the attributes they share ... Another repetitive activity occurs during the process of finding relationships. It becomes very difficult to determine domain relationships if no knowledge structure exists to which we can relate domain entities as we discover them. Finding relationships requires the ability to traverse (browse) a knowledge structure over and over to find how entities relate."

The author believes that support for these two activities "will greatly improve our capacity for performing DA. Domain information grows rapidly during the analysis process. Control and order of this information is, therefore, essential." He proposes additional research into conceptual clustering and additional derivation of tools. He also proposes to explore knowledge representation research and Hypertext technology, using existing tools to support management of information and knowledge structures.

In "Conceptual Modeling for Software Design" [GRE88],Greenspan and Feblowitz emphasize the importance of fully understanding the problem domain, not just the functions performed by an application.

"It stands to reason that a system (or a tool or environment) that automates some aspect of software design must contain knowledge of the application domain, if it is to perform intelligently. Some of the issues that need to be addressed are the following:

- o Which domain knowledge to capture
- o How to represent it and organize it
- o How to acquire and maintain it
- o How to use it to perform some software design task (significantly better than were the domain knowledge not present)"

The authors used an object-oriented knowledge representation framework to represent domain knowledge. They believe that reusers need domain-specific tools to provide an effective design environment, but that a domain-independent **conceptual modeling platform** (CMP) is also needed. This platform would form the basis of domain-specific representations and tools.

The particular project the authors worked on involved designing telecommunications network services. In this project, domain knowledge includes the environment (user and network) with which the service interacts, the software components from which the services are composed, and the services themselves.

"Our current work involves the crafting of appropriate knowledge representation and acquisition tools for defining network services. We are concerned with ... what should be the

capabilities of a software/services creation environment for service definition. Such an environment will involve languages, tools, and techniques for specifying, evaluating, and evolving network services ...

"We view the service creation environment of the future as containing a body of domain knowledge about the network services domain that is relatively stable ... and a set of tools that assist in rapidly composing new services. The main conjecture that is implicit in this approach is that composing services will be much easier, given the presence of the domain knowledge."

Figure C.5-6 depicts services constructed from components that are defined in terms of the domain model. The domain model consists of a set of interacting objects. "Services are defined ultimately by reference to the effects they prescribe on the model ... The components themselves are defined in terms of the domain model, but services are composed only from components (and other services)."

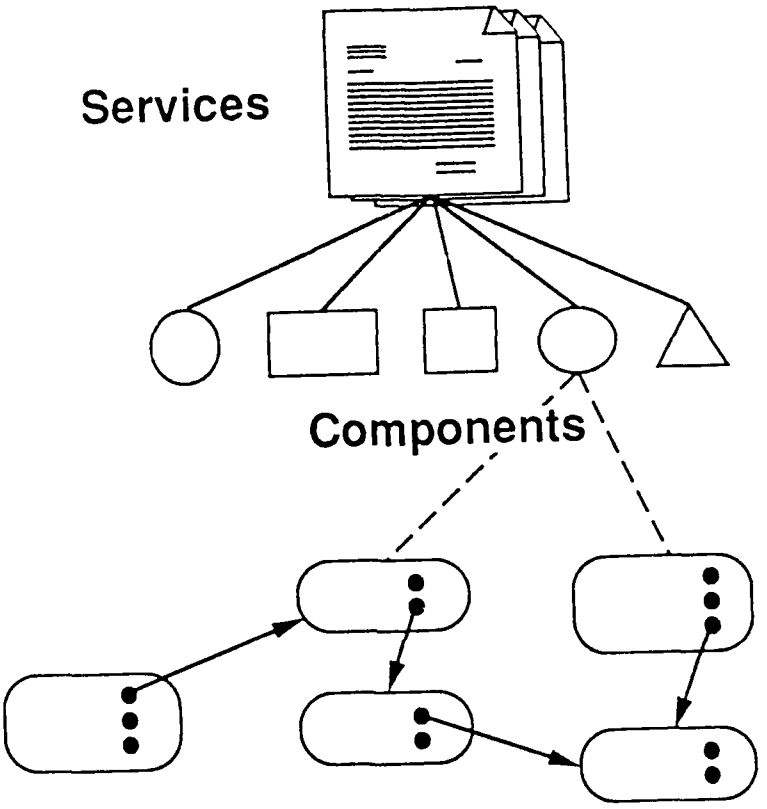
Returning to the conceptual modeling platform (CMP), the authors state that the CMP "is needed to provide more generic modeling concepts on top of which domain-specific platforms can be built for several domains. A domain-specific platform for some subdomain of network services deals with such concepts as customer, calls, and equipment failures, and provides specialized editors, display techniques, dialogue control and other tools for these concepts, whereas the conceptual modeling level deals with entities, activities, assertions, etc., which encapsulate generic specification techniques in a uniform, hybred model."

Naturally, this same concept could be applied to other development environments and other domains. A summary of the author's main points follows:

1. "Understanding and stating software requirements should be done using an explicit domain model.

2. An essential design activity -- composing software from components -- requires domain analysis to craft a good set of building elements and composition techniques.

3. Domain-specific platforms should be created for software in specific domains, but a conceptual modeling platform is needed to support the construction of these platforms."



Domain Model

Figure C.5-6. Domain Model

C.6 IBM

McCain of IBM Federal Systems Division proposes the following use of domain analysis in the process of decomposing a problem solution (a set of component requirements) into reusable subcomponents [MCC85]:

1. Perform a domain analysis prior to implementing the component. Identify potential users of the software and their specific needs, with emphasis given to identifying commonality across the domain under consideration. Conduct a review to influence the implementation.

2. Determine if existing software meets the requirements of the component. If so, reuse it. If not, proceed with the development. If only partially, define a new component specification that reflects the requirements not accommodated and repeat step 1.

3. Identify reusable objects applicable to the domain under consideration (including their associated reusable operations).

4. Determine reusable abstractions applicable to the problem solution. The domain analysis should exert significant influence on these definitions. Layering should be used to partition abstractions to achieve different levels of reusability potential.

5. Define a formal interface specification for each reusable abstraction from step 4 which includes an explicit definition of *all* interfaces associated with usage of the abstraction.

6. Define any usage and/or implementation constraints on the abstraction, with each constraint accompanied by rationale. Based on an analysis of the constraints, the abstract interface from step 5 may be redefined.

7. Implement the components according to the interface specification. Conduct a review to validate the implementation.

Validation via review is a key aspect of the methodology. McCain recommends review from at least three points of view to ensure that reusability and maintainability objectives are met:

1. Domain Analyst - identifies areas of commonality and existing related software

2. Software Component Engineer - identifies ways to achieve optimum reuse and reduce maintenance costs

3. Component User - determines if the components are usable and meets its requirements

C.7 Knowledge Systems Corporation

Auer and Adams in [AUE88] believe that "there is almost never a problem whose domain definition does not change once the initial domain model has been created." "Therefore, when defining a domain model, we should assume that the definition of the domain will change, and take precautions to ensure that changes will minimally impact the model."

They propose several steps to minimize the impact of those changes. A difficult, but critical, task is to "identify and categorize the entities and, if at all possible, limit each entity to one category." The authors prefer a bottom-up approach for entity identification and categorization, but feel that a top-down approach could be used as well.

"The first step is to identify 'independent' objects (classes). These objects know only about their own physical properties and are totally self-sufficient ... Examples of objects in the library domain which should be independent are Books and Bookshelves."

"The second step is to identify 'functional' objects (classes) and their categories. These objects actually manipulate the independent objects ... in order to reach some greater end. Objects that will be used by an application to get information should be functional objects." The authors note the importance of drawing a boundary between the domain and the application, which will "most likely end up at a neutral zone, as it will be difficult at times t define where the domain model stops and where the application begins."

"The next step is to identify 'supervisory' objects (instances). These objects sole purpose is to supervise the independent objects," keeping track of where the objects are at all times. "Once all these [supervisory] objects are defined, the function and scope of each should become relatively apparent (and therefore they can be bound to a class). Some will fit nicely into the functional objects, other will need to be shared among a small number of the functional objects, and still others will be needed by many functional objects. Problems will sometimes arise due to the overlap of some of these objects." Techniques for subsetting and supersetting the objects are used to simplify change management in such objects.

"Last, the definition of the functional objects should be completed. The scope of the supervisory objects should play a major role in helping to identify the functional objects which should be integrated into one collective functional object (which would contain the supervisory object). When duplicate functionality is found ... abstractions and support objects should be defined. It is of utmost importance, when defining any object, to keep the scope of the object small, and not to duplicate functionality across objects or across [operations] within an object."

While the authors admit there is "no way of guaranteeing the design of any domain model will be optimal, " the use of "encapsulation and scope limitation ... can reasonably ensure that a domain model will evolve smoothly as knowledge and insight is gained."

C.8 McDonnell Douglas Astronautics Company

The most significant work in the area of domain analysis for reusable software for DoD Ada applications is the CAMP effort, which focuses on reusable software for the missile applications domain (as one of many classes of real-time embedded systems). Volume I of the CAMP study results [MCN86a] summarizes their experience in performing a domain analysis for this application area.

The authors point out that their study would have been impossible without an in-depth domain analysis, but note that "[f]ew organizations or researchers have conducted an in-depth domain analysis of real-time software systems. The primary reason for this seems to be that investigators were put off both by the complexity of the software and the application expertise required."

The CAMP study defined the following three steps for performing a domain analysis (addressed in more detail below):

1. **Domain definition** - "the process of determining the scope of the domain analysis (i.e., what application areas will be examined)."

2. **Domain representation** - "involves the selection of a set of applications which characterize the domain under investigation."

3. **Commonality study** - identifies operations, objects, and structures common to all or most of the applications under examination, which become "candidates for construction as reusable software parts."

The CAMP report cautions that although **domain definition** "appears to be a straight-forward task, there are several factors which introduce complexity: [fuzzy domain boundaries, domain overlaps and domain intersections]."

Because domain boundaries are not rigid, and areas of contention must be identified early in the analysis so that a domain's functional areas can be clearly defined.

Overlapping domains occur when some areas in the domain of interest also fall within another domain. As such, the CAMP developers found it can be easy to get sidetracked into the overlapped domain.

Domain intersection "occurs when a functional area within the domain under examination applies to a wide spectrum of domains. Figure C.8-1 illustrates example intersections between application-specific (vertical) and application-independent (horizontal) domains. Getting sidetracked can also occur in the case of domain intersection.

CAMP relied heavily on management control and reviews to ensure the domain analysis was limited to the domain of interest. The authors note that domain overlaps and intersections only become a "problem" when resources for performing the analysis are limited, as was the case with CAMP.

The application selection process for determining the **domain representation** must deal with the availability of information about the application, the quality of that information, and the representative of the application to the domain.

The availability of documentation for the applications was very important to the success of CAMP. Representation of all aspects of the domain was found to be critical. However, existing applications are often found to be lacking documentation, or their documentation will be lacking critical information. Whether the application is representative of the domain is the key factor and the one over which the analyst has the most control. Time and personnel constraints will prevent the study of all applications within the domain of interest. The CAMP contract required the selection of ten missile systems to perform the commonality study, and care was taken to ensure that these ten were indeed representative of the domain.

The CAMP effort found that the **commonality study** "is very similar to the analysis performed in the construction of an expert system. Both these analyses are attempts to formalize a body of knowledge... In the case of a reusability domain analysis the knowledge is needed to [distil] abstract requirements from concrete instances of requirements."

Software documents such as requirements specifications, design specifications and, as a last resort, code listings provided the raw data for the analysis. CAMP researchers found that requirements specifications alone were not sufficient to identify potential software parts, and design specifications played an important role. This was found to be especially true in addressing internal functions that don't map directly to any external requirement. Code listings were found to help the researcher understand the specifications.

The CAMP team found the functional strip method to be the best for actually performing the commonality study, where one analyst examines a particular function across all domains in the set, to be the best for actually performing the commonality study. The narrowness of the strips influenced the ease and probability of finding commonalities. Narrowing the strips made it easier to recognize similarities, but narrowing the strips too far could make it impossible to recognize higher level commonalities. Data flow modeling was also used to highlight similarities between the functions.

"Two barriers which need to be overcome in this analysis are arbitrary differences and notational differences." Both problems proved to be major hindrances to detecting commonality, and required both application and software engineering expertise to resolve.

"The presence of multiple levels of commonality has a major impact on the way in which a domain analysis is performed." CAMP identified three levels of commonality that domain analysts must be sensitive to the following: functional commonality, pattern commonality and architectural commonality.

> **Functional commonality** "involves a black box view of common operations... [and] is the type of commonality classically associated with software reusability."
>
> **Pattern commonality** involves recurring logic patterns that are more complex than simple black box functions. A finite state machine is a good example of pattern commonality.
>
> **Architectural commonality** is similar to pattern commonality but on a larger scale. It involves common models of major components of the domain's software systems. "One clue that architectural commonality exists is the existence of standard pictures of a process in text books on the subject in question."

CAMP identified several characteristics of the missile domain which affected the detailed domain analysis:

- o Very high degree of data flow interconnectivity
- o Complex decision-making
- o Large number of mathematical data transformations
- o Little data movement
- o Relatively simple data structures
- o External interfaces with special-purpose equipment
- o Processes that have rigid temporal relationships
- o High use of intermediate results of calculations
- o Time-driven processes

These characteristics had an impact on what parts were designed, how they were designed and how they fit together. For example, the use of non-parameter procedure communication and data update protection led to special data packages and communication mechanisms. A heavy percentage of decision-making functions led to macroscopic analyses that resulted in the identification of pattern components. Heavy use of mathematical transformations led to identification of standard primitives. Because few complex data structures were used, no time was spent analyzing commonality among data movement operations and structures. Requirements for peripheral interfaces led to the identification of standard interfaces. Asynchronous behavior required analysis of the use of Ada tasking; this resulted in multiple categories of task shell components and a controller component. Heavy use of intermediate results required building parts where the data and the calculations are in separate packages. The many time-driven actions in the domain led to the capture of timing paradigms in process sequencer components.

Once the parts have been identified, they must be implemented in Ada. "There are three main issues surrounding the specification of parts within the CAMP domain: what constitutes a part, how the part will be used, and how to specify a part so that it is reusable." The part specifications included a description of the functionality provided by the part (inputs-processing-outputs), performance of the part, and the environment or context of the part. Desirable characteristics of reusable CAMP parts included well-defined interfaces, simplicity, and flexibility. Six design methods were applied to guide the part development: typeless, overloaded, generic, state machine, abstract data type, and skeletal. Two modes of part usage were developed: bundled and unbundled. Bundled components are provided to the user as Ada packages, and are stored and retrieved from an Ada library system. Unbundled components permit access to code fragments which may be incorporated into newly developed Ada code. Management of unbundled components is provided by a "text library."

Because it is likely that a large number of parts will be identified during the analysis, CAMP recommended the development of a Software Parts Taxonomy to help domain analysts organize their work. The CAMP domain analysis resulted in the identification of over 200 parts.

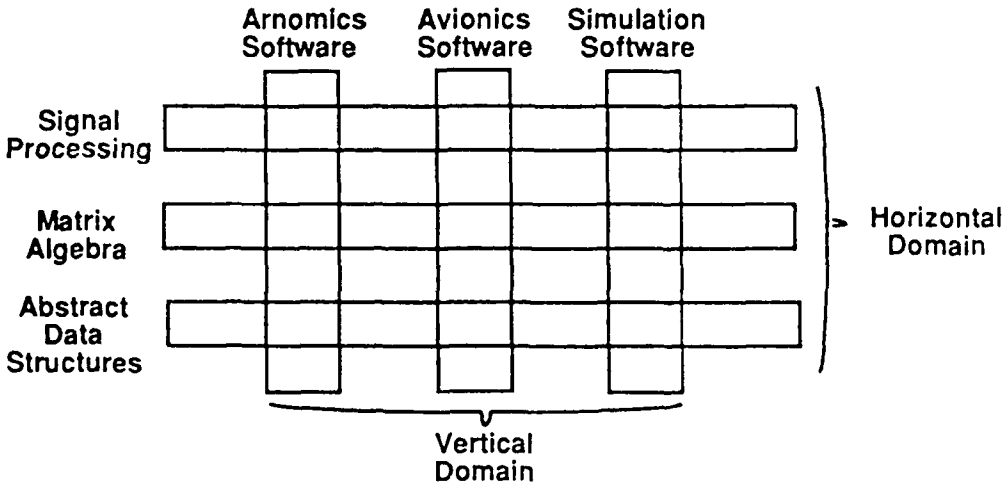
As parts were identified, they were additionally classified according to how they could be implemented: simple, generic or schematic. Simple parts intended for reuse "as is". Generic parts are parts which are tailored through instantiation, an Ada language capability for parametrization of a unit's data type and externally referenced subprograms. Schematic parts "consists of a

template and a set of construction rules which are used to generate a number of specific components." A schematic part is somewhat like a standard design, except for the automated construction.

The resultant CAMP parts were also partitioned into two classes according to whether they were domain-dependent (applied only to the domain of interest), or domain-independent (useful in other application domains while still being highly relevant to the domain of interest, such as math parts).

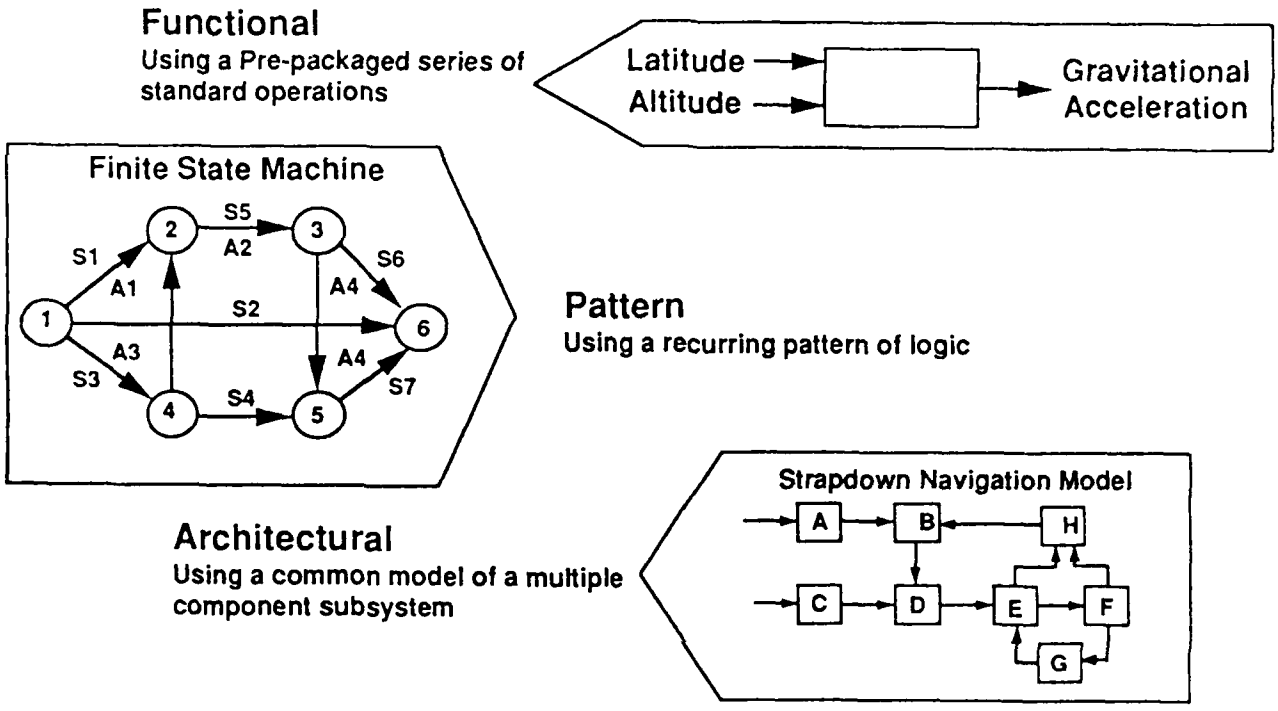
Volume II of the CAMP study report [MCN86b] addresses the development of the CAMP software parts catalog and an automated software parts composition system. A key aspect of their automation research was in the applicability of expert systems technology, where the researchers concluded that "[expert] systems have a high potential in the automation of the software parts engineering process," although several major drawbacks were identified.

The resulting parts composition system, Automated Missile Parts Engineering Expert (AMPEE) system, included application domain-specific knowledge to reduce the amount of human effort required in specifying the parts and to produce more efficient parts. The user specifies the components to be constructed via a domain-dependent dialogue.



Vertical and Horizontal Domains

Figure C.8-1 Vertical and Horizontal Domains



Three levels of commonality

Figure C.8-2 Potential Levels of Commonality

C.9 Microelectronics and Computer Technology Corporation

Biggerstaff addresses the content and role of domain models in [BIG88]. In this paper, he identifies four kinds of information to represent a domain model as illustrated in Table C.9-1, Level of Abstraction in Domain Model, and described below.

- o Object-oriented classes - "strong operational capabilities but ... weak in its ability to describe ... in informal, human oriented terms ... Another weakness that it has is its target program specific nature."

- o Software Engineering Design (SED) representations - abstract "to present a more general but still operational description." However, "it is [still] specific to a single implementation."

- o Generalized SED (GSED) representations - attempt "to overcome this [implementation-specificity] weakness by further abstracting the [SED] representation."

- o Conceptual abstractions - describe objects in "human-oriented terms." "[T]his representation allows far greater flexibility in the kinds of structures that are acceptable to describe a given concept as well as the level of commitment to their actual existence."

All of these kinds of information may be used to describe different aspects of a single object, varying "along several dimensions -- the properties of their representational forms, their reason for existence, and their behaviors." The author argues that multiple representations of the same object are required because "each representational form has a differing role, a differing set of capabilities."

"These representations can be used to complement each other in the course of working with and understanding a program. In general, the more operational and detailed the question that a Software Engineer wants to ask, the more operational the representation chosen to help. The more the question is oriented to the computational purpose or objective, the more one must move toward the Conceptual Abstraction representation."

"Alternately, operational representations may create inconsistencies with the descriptive representations, and their resolution will lead to improved understanding of the target entity." Descriptive representations may lead to finding actual artifacts, but they may be "out of synchronization with some subset of expectations." The user must then use techniques (such as symbolic execution of code) to better understand the artifacts, probably revising the user's original expectations in the process.

"Currently, we are building a model of a multitasking window system domain to be used in a Design Recovery system called DESIRE ... This domain model is used to help a Software Engineer interpret and understand code from the domain ... Based on information in the model, DESIRE reads the code and tries to recognize expected structures in the code. It presents candidate structures to the Software Engineer for further analysis and verification."

Table C.9-1 Level of Abstraction in Domain Mode

Level	Purpose	Properties
Code (e.g., object oriented class)	• Execution	• Implementation specific • Conctrained mostly by programming language • Formal objects • Operational form
Software Engineering Design	• Abstract away Detail • Weakly relate to informal concepts	• Implemetation specific • Constrained by mix of program language and domain • Semi-formal objects • Abstractly operational • Reduced detail
Generalized *Software* Engineering Design	• Abstract away detail • *Weakly relate to* informal concepts	• Not implementation specific • *Constrained by mix of programming language* & domain
Conceptual Abstraction	• Strongly relate to informal concepts	• Not implememtation specific • Object like structure • Non-operational (i.e., prescriptive form)

Level of Abstraction in Domain Mode

Built using a hypertext system, DESIRE represents program information in terms of hypertext webs "that can be examined and navigated by the Software Engineer". Inputs to DESIRE includes the domain model, program code, design and requirements documentation, and the "designer's understanding and interpretation". Outputs from DESIRE are a reuse library and "artifacts to aid understanding." Using the current version of DESIRE, the engineer must apply domain knowledge manually when analyzing the code.

"The key property of the model is that [it] makes strong use of informal or semi-formal information in the interpretation process and relies on fuzzy matches to find candidate abstractions. It is the belief of the author that such informal or semi-formal information plays a far greater role in the Software Engineer's ability to understanding programs than we have heretofore recognized. In fact, it may be the single most important ability of the human worker that makes the understanding and interpretation of large complex programs even possible. Without using informal information to provide clues and expectations, it might be virtually impossible to understand such programs."

C.10 University of Texas - Austin

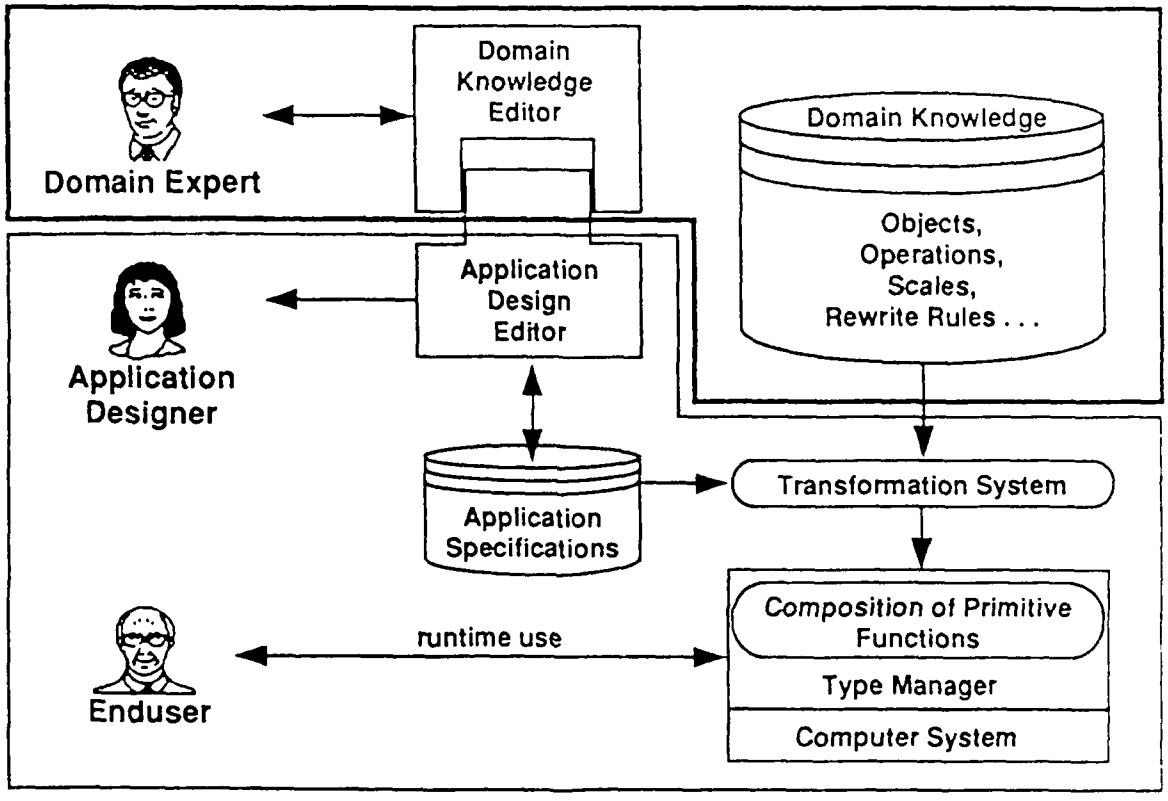
Iscoe addresses the problem of modeling application domains in the content of automatic program generation in [ISC88]. "Although most would agree that a combination of programming knowledge and application knowledge is required to automate the construction of application programs, there is not yet agreement as to how to model an application domain. The problem is to create a model for domain knowledge that is general enough to be instantiated in several domains and expressive enough to specify and generate application programs. This model will then be used to build a meta-generation system that when instantiated for a particular domain can automate some of the aspects of application programming for that domain."

The author is involved in the creation of a system (called Ozym) to support this process. The effort will include an object-oriented model for representing the knowledge (partially formalized), portions of a methodology, and algorithms and techniques for leveraging characteristics of domain knowledge in generalizing across domains. "Scaling theory, a well defined mathematical system used for classification in other fields, forms the basis to characterize real world object attributes. Scales are the major mechanism used to retain the application knowledge usually lost in the mapping from user specification to hardware implementation." The author then plans to define a model and instantiate at least two domains and analyze the results. Figure C.10-1, Iscoe's Ozym System, shows how the proposed system would work.

Iscoe begins by justifying the need for formal domain models. "The lack of a formal model to represent information at the application domain level results in a severe information loss during the mapping process as it progresses from requirements and specifications expressed in terms of an application domain through computer science and programming language domains to a final level of implementation" [see Figure C.10-2, Mapping Through Domain]. He notes that "[m]ost real world domains are neither well defined nor clearly bounded and consequently large amounts of effort are required to elicit and clarify the users image of, and specifications for, an application program ... Software designers implicitly use and reuse application domain knowledge to prevent specification errors, and it is the absence of this knowledge that allows many types of errors to occur."

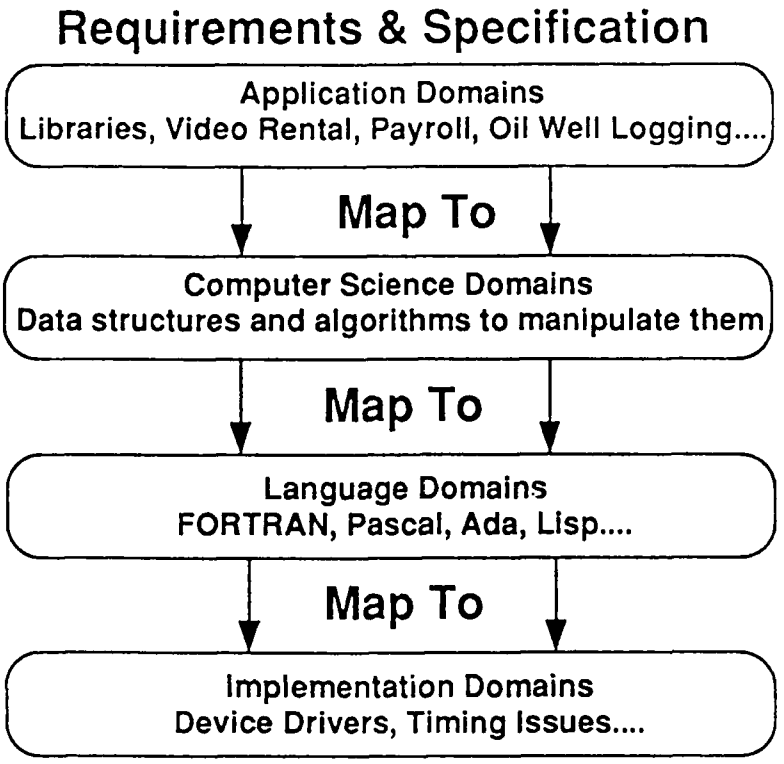
Iscoe proposes these following steps for the research:

1. Create a model of domain knowledge.
2. Implement the model.
3. Instantiate the system for the ... domain.
4. Specify and generate programs within the domain.
5. Instantiate the system for another related domain.
6. Refine the Model.
7. Compare the instantiations.
8. Identifications of characteristics and traits that generalize across domains.
9. Identification of algorithms and techniques that can be used across a class of domains.



System Overview

Figure C.10-1 Iscoe's Ozym System Overview



Mapping Through Domain

Figure C.10-2 Mapping Through Domain

"The model must allow for knowledge to be represented in a way that is understandable to the domain experts, the users, and to the computer. Informally, the model consists of objects and the operations performed upon those objects within a particular application domain ... Our model adds domain knowledge to the standard object-oriented model by formalizing an application domain's natural scales ... and incorporating them into definitions of objects and operations." Iscoe's model also results in "a new approach towards solving some of the problems of multiple inheritance."

"Scaling theory is a methodology with a mathematical basis ... used in our model as a means for classifying object attributes in a manner that is both formal and yet captures enough of the underlying real world semantics that the resultant model can be used for specification and transformation purposes." The following hierarchy of scales is proposed by Iscoe for use in object modeling:

- o *Nominal scales* - " ... group items by placing them into equivalence classes ... Domain semantics are captured by creating classes in such a way that items are as homogeneous as possible within an equivalence class and as heterogeneous as possible between equivalence classes. Proportions, percentages, and ratios are the important statistical characterizations of nominally scaled items and can be trivially calculated for any finite set."

- o *Ordinal scales* - " ... group items by ranking them according to the amount to which they possess some characteristic defined by the scale. Ordinal scales are nominal scales for which a strict ordering has been obtained." Partial orderings are also sometimes useful. The rankings "may be obtained in a variety of ways," but ordinal scales have no information about the magnitude of differences between groups.

- o *Interval scales* - " ... ordinal scales that group items according to their position on a scale of standardized intervals or units." Interval scales allow comparisons between classes and calculations such as averaging.

- o *Ratio scales* - " ... interval scales that have an absolute, non-arbitrary zero point. Even more mathematical operations are possible with ratio scales. "Ratio scales are the most powerful scale for measuring the properties of objects."

In Figure C.10-3, the author illustrates the four scales, their relationships, and "some of the ways that the careful characterization of attributes can be used." "Although scaling theory only gives a foundation of measurement and does not prescribe the proper way to assign properties to objects, it leads naturally to a methodology that can be used by a domain expert who is aware of the *natural* classifications of objects within domains."

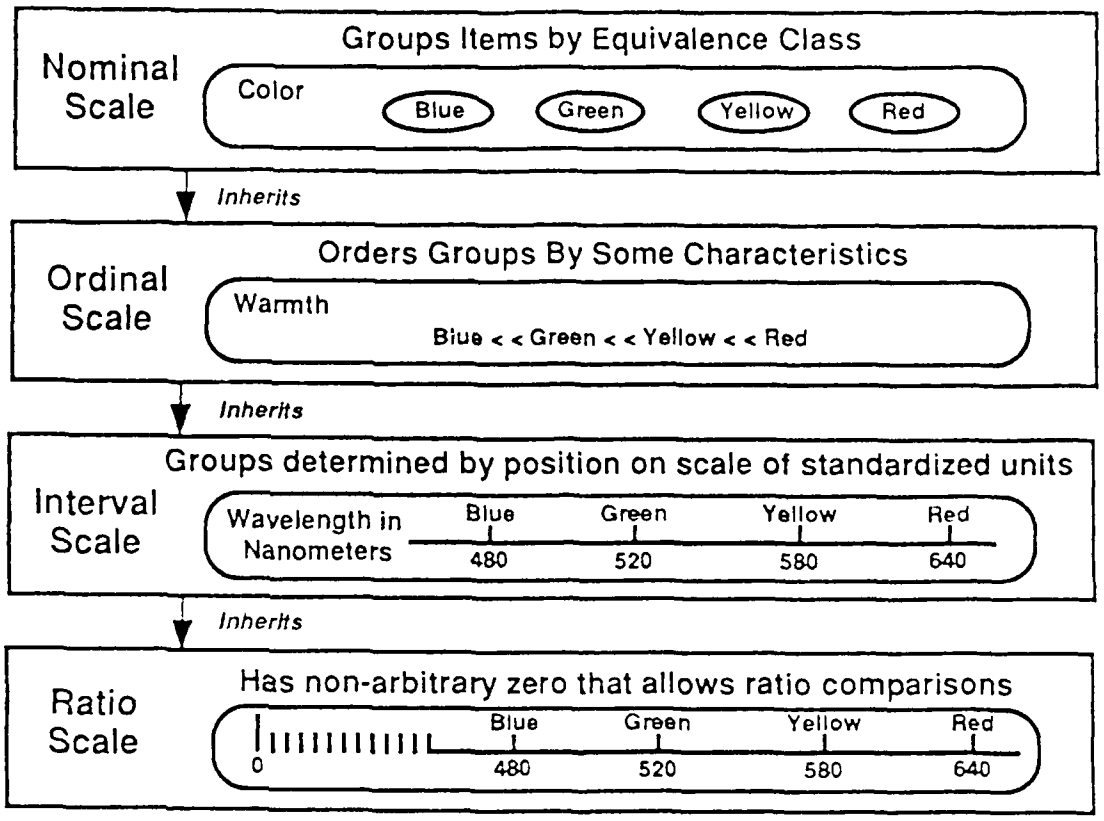
Iscoe describes in some detail the four areas where scales are helpful:

- o Defining and understanding state transitions.

- o Characterizing objects with population parameters.

- o Efficiently implementing runtime code.

- o Communicating with the enduser.



Scale Hierachy

Figure C.10-3 Scale Hierarchy

He continues with a discussion of how scaling theory can assist in classifying and characterizing domain objects. "The first step in understanding a domain is recognizing the domain's natural objects. This is a view that is advocated in different ways by Jackson, Neighbors, Booch, and Prieto-Diaz. In a domain-specific view of the world, classification is determined by context. This can be formalized by classifying the properties and operations of an object in terms of domain-specific scales."

"Implicit in the classification of objects through measurement of their properties is the idea that object classification with the appropriate units and scales is a domain-specific activity." The author suggests that there may be "fundamental properties of objects that are not domain-specific, but are independent across domains ... From a practical standpoint ... certain objects properties appear regularly across classes of domains. Ideally, the model will be able to describe a number of different domains by accumulating a set of fundamental scales for domain classes in the manner that physicists have developed their models in terms of only a few basic scales."

"[T]he general problem of application domain modeling requires another concept--the notions of measurement and observational granularity. *Measurement granularity* is the level of granularity at which an item can be measured and is largely dependent on the available technology, while *observational granularity* is operationally defined as the 'normal' level of granularity for an item as it is usually observed, or recorded, in a domain." Iscoe proposes some rules for exploiting measurement and observational granularity by domain experts: "*neither original nor derived information should ever be expressed at a finer level of granularity than is justified by its component parts, the algorithms used to compute it, or by its original level of measurement granularity.*"

"Domain knowledge in the form of scales can help to solve the problem of resolving conflicts and other ambiguities of multiple inheritance." For example, "composing the properties of two objects into a single object with its own set of properties is not a problem that can be solved by simply taking the union of the properties of the two objects from which the resultant object is derived." Domain-specific information about the meaning and application of scales is required to address the problem. "We believe that many composition rules can be formalized in terms of scales, and the some of these rules will generalize across domains."

C.11 MCC--Lubars

Lubar's paper describes the IDeA design environment, which provides assistance during the construction of software designs, in particular, supporting the reuse of abstract software designs. These designs are represented in the form of design schemas. "Before IDeA can provide support for deign reuse, a designer experienced in certain application domains must populate IDeA's design reuse library with the schemas for those new domains. This is an extremely tedious and difficult process," and currently entirely manual.

Lubars makes a distinction "between the processes of analyzing an application domain for reusability and the actual construction of the reusable artifacts for that domain. We term the former *domain analysis* and the latter *domain engineering* ... In performing a domain analysis for IDeA, that analyst must consider the boundaries of the domain in terms of what objects to include an to what degree they should be abstracted."

The author identified a tradeoff between development effort and subsequent reusability, where "abstraction enhances an object's reuse [potential], but is a difficult and expensive process ... Once a domain analysis is complete, the domain must be engineered to support reuse. The engineer must decide which design schema and other reusable artifacts must be constructed to support design reuse in that domain."

IDeA supports several types of reusable information, stored in knowledge bases:

1. "Properties about objects in the domain. These are attributes that distinguish the objects from the other object types in the domain ... Properties are organized into a collection of trees, where properties higher up in the tree are abstractions of properties beneath them and sibling properties are disjoint from each other... These property trees are helpful in classifying data type information and are similar to the concept of classifying software using *facets*."

2. "Data types are described in terms of their properties and are organized into a type lattice ... Any subset of those properties characterizes a more abstract data type ... The data type lattice, along with the data properties, provides a vocabulary with which uses can communicate their specifications to IDeA. The data type lattice is also used in classifying and selecting design schemas."

3. "Design schemas correspond to abstract design solutions for a class of related design problems. They may be viewed as design idioms and they frequently correspond to domain-oriented operations." The schemas may represent various levels of abstraction, with "additional detail determined through design specialization and refinement ... Design schemas in IDeA represent abstract software design solutions in the form of dataflow diagrams."

4. "Schema specialization rules are mappings between an abstract design schema and a more specialized design schema ... Schema specializations are almost always associated with the addition of requirements or design commitments beyond those implied by the more abstract design schema. Alternative specialization rules frequently represent a choice between alternative requirements or design commitments and frequently result in derivations that lead to different design refinements (implementations)." "[S]chema specializations distinguish between different classes of design solutions and are selected based on one or more user commitments that identify the desired solution."

5. "Schema refinement rules are mappings between a design schema and a dataflow design that represents the refinement or implementation of that schema. Refinement rules do not change the level of abstraction."

6. "Type constraints are included in design schemas to propagate property assignments of data types to other data types that share the same abstract property class."

"Domain analysis in IDeA is viewed as the process of analyzing an application domain to determine the operations, data objects, properties, and abstractions that are appropriate for designing solutions to problems in the domain. Domain engineering for IDeA is the process of constructing specific design schemas, property hierarchies, constraints, a data type lattice, and specialization and refinement rules for enabling IDeA to assist in the construction of designs in that application domain. Thus, domain analysis may be viewed as focusing on key aspects of the application domain without particular consideration to the engineering for design support. In contrast, domain engineering is primarily concerned with organizing the concepts identified during domain analysis in order to construct the specific artifacts that IDeA represents in its knowledge base."

"Domain analysis in the IDeA framework generally consists of the following categories and steps.

Analysis of similar problem solutions:

1. Identify the domain-oriented operations and data objects for solving a particular problem in the application domain.

2. Extend the set of domain-oriented operations and data objects by considering similar problem solutions in the application domain.

3. Group related domain-oriented operations and data objects into common abstractions, as identified in steps 1 and 2.

Repeat steps 2 and 3 as much as necessary to characterize the particular problem solution class in the application domain.

Analysis of solutions in an application domain:

4. Extend the set of domain-oriented operations and data objects by considering other types of problem solutions within the application domain.

5. Group related domain-oriented operations and data objects into common abstractions, as identified in steps 1-4.

Repeat steps 4 and 5 as much as necessary to characterize the particular application domain.

Analysis of an abstract application domain:

6. Extend the set of domain-oriented operations and data objects by considering other problem solutions in related application-domains.

7. Group related domain-oriented operations and data objects into common abstractions, as identified in steps 1-7.

Repeat steps 6 and 7 as much as necessary to characterize the abstract class of related application domains."

The author compares step 1 with basic systems analysis. Steps 2 and 3 "provide a better characterization of the application domain and ensure that the subsequent domain engineering will produce components that are more broadly applicable." Steps 4 and 5 "ensure that the resulting knowledge-base will be applicable to multiple applications within the application domain." Steps 6 and 7 "provide design abstractions that are reusable across application domains." The difficulty and effort required increases with each step performed.

Domain engineering in the IDeA framework consists of the following steps.

1. Assemble the identified data objects into a type hierarchy.

2. Identify properties that distinguish between the various data objects in the type hierarchy, and organized those properties into property types.

3. Define mappings between the data types and the sets of properties that distinguish them. These mappings locate the data types in the IDeA type lattice.

4. Assign abstracted data type names to other interesting combinations of data properties.

5. Construct design schemas for each identified domain operation, including schemas for each abstraction of domain operations.

6. For each design schema, assign data types to the input and output dataflows. For the abstract design schemas, assure that the data types are abstractions of the corresponding data types of the more specialized schemas.

7. Assign type constraint variables to properties of the different data types that must match in each of the design schemas.

8. Create specialization rules between the abstract design schemas and each of their more specialized design schemas.

9. Create design refinements that implement the specialized design schemas in terms of more detailed dataflow diagrams. As in steps 6 and 7, assign data types and type constraint variables to the dataflows in the refinements."

The most difficult design engineering task is organizing the property trees and identifying the type constraint variables. The tasks become more essential as the level of abstraction is raised, because more applications are covered. Also, more information is required to distinguish among the different designs and constrain their applicability.

The author reiterates the trade-off between the amount of abstraction and the amount of effort to define the domain. "Beyond a certain point, additional abstraction does not provide enough coverage of desired problem solutions to justify that additional expense in constructing the abstractions." He recommends performing more separate domain analyses over trying to abstract to a single domain, although some domain information might not get covered. "The cost of developing the ... applications that were not completely covered by the knowledge base might be less than the additional cost in further abstracting the knowledge base to provide the coverage." Lubars also notes that smaller domains generally have more common constant attributes, reducing the need to manage and select from among alternative attributes.

"There is little data to provide good estimates regarding how much domains should be abstracted and under what circumstances additional abstraction actually pays off in terms of increasing the reuse potential of a domain." The author considers clustering moderately related domains under a single abstraction risky. However, the effort does pay off in the case where the applications are very similar, because the effort to discover the abstractions is reduced. "The reduction of effort is derived from the reduced amount of work in identifying and categorizing distinguishing properties, in the engineering of type constraint variables for the resulting schemas, and in the construction of specialization rules between abstract and more specialized design schemas."

Lubars provides a few heuristics to help decide whether to further abstract a domain:

1. "[W]hen the fraction of similar components in two or more solutions is quite high relative to the total number of components (perhaps >80%)."

2. "[W]hen the abstractions of components ... differ in terms of only a few properties." He suggests an 80-20 rule, where 80 percent of the solution differences could be expressed in terms of 20 percent of property decisions.

3. "[W]hen the potential for reuse of schemas and other knowledge in the domain is high. This heuristic may modify the application of the other heuristics, since increased potential for reuse offers more potential to recover the additional cost of further domain abstraction."

C.12 Murray State University

McGregor proposes the use of object-oriented modeling and specification techniques for domain analysis. In [MCG88], he proposes the use of a class hierarchy to represent a domain, where a "class is a template for a relation." Ideally, he recommends a "one-to-one correspondence between inheritance structures and domains", as opposed to a single inheritance structure from which all classes are derived." According to this model, each root "is an abstract representation of the domain ... Successive layers of class definitions provide more specialized descriptions of members of the domain."

The author notes that "[r]elationships between objects in domain analysis can be handled in much the same way as they are handled in relational database design." However, certain relationships (e.g., is-a) are "difficult if not impossible to represent." McGregor used an extended relational model to support this operator. "By using a relational approach to domain representation, the entity-relationship modeling methodology can be used to guide the analysis process. This methodology provides guidelines for knowledge acquisition, documentation, and diagraming techniques. This approach also closely cooperates with object-oriented techniques for system implementation."

C.13 Notheastern University

Lieberherr and Riel address some of the issues concerning domain analysis raised by the Domain Analysis workshop at OOPSLA 88. [LIE88]

They define a domain model to be "a collection of parameterized languages and their semantics. The parameterized languages define simultaneously the object which belong to the domain model and a parametric collection of languages for describing those objects. The semantics are defined in a syntax directed way at four increasingly more specific levels ..." **Protocols** provide names of the functions. **Signatures** provide argument and return types. **Abstract behaviors** provide implementation-independent function descriptions. **Concrete behaviors** provide implementations of the functions using a language supporting object-oriented programming.

The authors recommend defining the domain model using EBNF-like parameterized productions. Then, the domain model is instantiated by instantiating the parameterized languages. Domain models can be combined using techniques from module interconnection languages.

With respect to object-oriented approaches, the authors find factoring common attributes and specialization of attributes the most useful aspects of inheritance to apply to domain models. They also believe that support for multiple inheritance is essential.

The remainder of the paper describes work at Northwestern University on Demeter™, "a meta-system for object-oriented programming and knowledge representation ... developed over the past two years."

Demeter employs EBNF-like productions in building a class dictionary of the legal objects and their associated operations. Their notation allows modularization of these productions to support structuring of large class dictionaries, and so is a form of module interconnection language. In addition to the class definitions, the class dictionary contains parsers, pretty printers, object editors, application skeletons, software development plans (providing directions on the best way to develop an application), and static and dynamic type checkers. They reuse off-the-shelf tools (such as yacc and lex) where possible.

The authors point out the lack of guidelines for object-oriented programming. In response to this lack, they developed the "Law of Demeter (or Law of Good Style)." "It basically states that messages be sent only to the direct instance variable objects and argument objects of a method. This constraint guarantees locality of types and has major implications for code modifiability. If an application satisfies the Law of Demeter, and its class dictionary is modified, we can guarantee that only a restricted set of methods need to be rewritten. If the code were written without restrictions then the modifications to the class dictionary could propagate throughout the entire application."

To date, Demeter has been used to implement "several sizable applications" including a language simulator and much of Demeter itself (about 14000 lines of Lisp/Flavors code).

C.14 OOPSLA '88 Domain Analysis Working Group

James Neighbors, the moderator of the Domain Analysis Working Group session at the OOPSLA '88 conference, documentedthe results of the session in [NEI88]. The report is summarized below. Note that it is organized according to a set of questions many participants had about domain analysis prior to the session.

Why Do Domain Analysis?

Productivity improvements in the construction of software systems from reusable components was the only reason cited.

Applicability of components to domains was defined as ranging from "very general (can be used in all systems) to very domain specific (can be used only in vertical applications specific to the domain)."

"*Domain analysis* is concerned with the process and results of identifying the domain specific components across the entire spectrum of applicable problem domains."

An organization should use the results of a domain analysis to do the following:

- o Check the specifications of new systems in the domain against the domain model
- o Educate people with the general structure and operation of systems in the domain
- o "Derive operational systems directly from the statement of the system in domain specific terms"

The Relationship Between Domains

They recommended that domain be arranged in a *domain hierarchy*, from the most general (the *execution domains*) to the most problem domain specific (the *application domains*). "To understand and perform the domain analysis of a very specific but complex domain such as EFT systems, we might enlist the previously performed analyses of banking and communication. Once a version of the analysis of EFT systems [was] done we would expect to provide implementations of [them] using specializations of the schemes which provide implementations for the constituent domains of the analysis, namely banking and communication."

How is Domain Analysis Done?

The working group attempted to apply their ideas of how to perform a domain analysis to an example problem domain, "library management systems". The second step was to apply the resultant domain to an example specification for a specific library system(not made available during the domain analysis). The group consisted of both *domain experts* and *domain analysts*.

The *domain experts* identified the *objects, operations and relationships* perceived to be important about the domain. "[D]ifferent members of the group used many different analysis techniques including: data-flow diagrams, entity-relationship diagrams, semantic nets, object diagrams, class hierarchies [with] inheritance." It was noted that most of the techniques used encouraged formation of the information into hierarchies, immediately presenting two problems: *depth of analysis* and *width of analysis*.

The *depth of analysis* problem is determining "how much support should be put into the library domain itself and how much should be required of the supporting domains." The *width of analysis* problem involved "a trade-off between specialization and generality", determining "is this function required by most of the systems built in this problem domain?"

"The members of the group seemed fluent in all the representations and it was generally accepted that there is no one favored representation for domain analysis. Each analysis technique can capture the 'flavor' of the particular domain under analysis from different overlapping perspectives. Usually only one each from the object hierarchy, data flow and control flow representations need be used to analyze a domain."

The group found that "*domain analysis rationalization* is one of the most important results of the process." This was despite many members initially having a *constructive point of view*, one which emphasized the ability to construct code from the specification. Understandability of the results, particularly communicating the issues and tradeoffs made during the analysis were viewed as very important. They suggested "a hypertext-oriented deliberation system such as MCC's gIbis system is probably the appropriate scheme for capturing the domain analysis rationalization. The nodes in such a system should contain analysis models as design artifacts and tradeoffs as issues and positions."

Applying the Domain Analysis

The specification of the example problem (taken from another conference on software specification) consisted of two parts: a set of required functions the system must perform and a set of required constraints on the functions. In comparing the specification to the domain model, "*the domain rationalization pointed out an error in a system specification in the domain far before any system design or construction was attempted.* Since the cost of correcting an error grows very rapidly as a system's lifecycle progresses, the ability to detect specification problems early may be a primary benefit of domain analysis."

Basic Domain Analysis Process

This section of the report summarized the working group's proposal for a "general process to apply to domain analysis.

1. Establish the domain subject area.
2. Collect the domain experts.
3. Establish the *depth of analysis* (see above).

4. Establish the *width of analysis* (see above).

5. Define the specific domain objects, relationships, and constraints.

6. Hand test the domain by attempting a description of a specific system in the domain.

7. Package the domain for constructive reusability by expressing it in a form for a transformational refinement tool such as Draco."

C.15 Reuse Tools and Enviromments Working Group

The Tools and Environments for Reuse working group formed at the Minnowbrook Conference on Reusability [RTE87] is currently investigating the automation of the reuse process. Their last meeting was held in Bass Harbor, Maine in June of 1988 [RTE88].

During the Minnowbrook workshop, the group addressed the use of domain analysis as part of the process for creating reusable components. Table C.15-1 Domain Analysis Activities and Support Tools, shows activities that were identified as part of the domain analysis process and a list of tools that could potentially be applied in supporting those activities. The numbers next to each tool indicate the group's assessment of tool class's maturity.

During the Bass Harbor workshop, the group reviewed the domain analysis process model developed by Prieto-Diaz, and developed the modified top-level data flow representation illustrated in Figure C.15-1. One of the group's major criticisms of current research into domain reuse is that only the development of the reusable products is addressed. The other part of the problem, actually reusing those products in new application systems, also needs to be addressed. The group defined the top-level data flow representation of Figure C.15-2 to illustrate this half of the reuse process. The group also criticized the lack of attention paid to the feedback of information from new systems development back into domain maintenance.

The group also analyzed the kinds of personnel involved in reuse and what viewpoints or organizations these people might promote or represent. Table C.15-2, Competing/Cooperating Reuse Interests, illustrates the group's conclusion -- that there are two main viewpoints involved in the reuse of software, those who are promoting the interests of reusable software (domain) developers, and those who are promoting the interests of end (application) system developers.

These can be both competing and cooperating interests. They are competing because the software qualities which can make a component reusable are often at odds with the software qualities meeting specific application requirements (e.g., performance). They are cooperating because the reuse of existing components makes the development of application software more productive, and the resulting applications provide input to the development of new reusable components.

On both sides, two major user roles were identified: those who are involved in developing new software (domain developers and system developers), and those who are involved in adapting existing software to fit their interests (system refiners and domain integrators).

An analysis of issues in domain evolution and maintencance was initiated as a result of NASA's experience in attempting to develop baselines of reusable software. According to Sid Bailin of Computer Technology Associates, NASA has developed two sets of reusable software components to date, with seven systems using both baselines. NASA is currently working on a third reuse baseline.

The first baseline failed because of incorrect resource assumptions and changes in their data networks (who gets what information and where it goes). The second baseline may fail because of the addition of distributed operations. For example, scientists now communicate directly with payloads rather than through a centralized, controlled system.

Table C.15-1 Domain Analysis Activities and Support Tools

SUBACTIVITIES	PARTIAL LIST OF POTENTIAL TOOLS
• Knowledge extraction	• Knowledge extraction tools (e.g., expert system building tools) [2]
• Identification of objects and operations	• Entity-relationship diagramming tools [2]
• Abstraction/relationships	• Object-oriented approaches and tools [3]
• Classification/taxonomy	• Semantic clustering tools, automatic classification
• Domain language, synthesis (formal/informal)	• CASE tools [2] • Parsing technology, BNF notation [1]

[1] = Mature [2] = Developed, but unproven [3] = Undeveloped

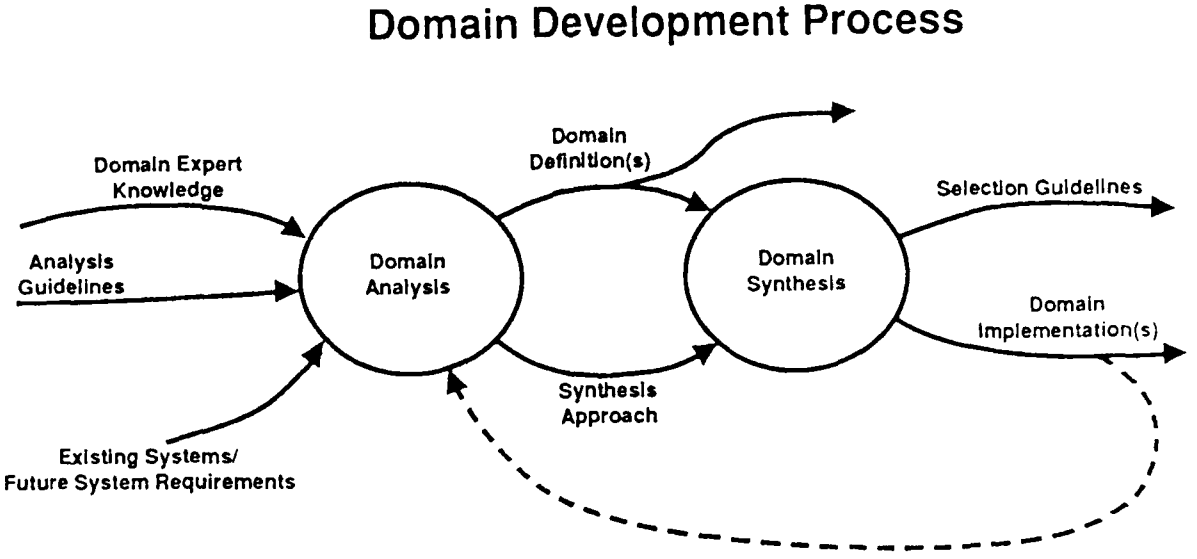


Figure C.15-1 Domain Development Process

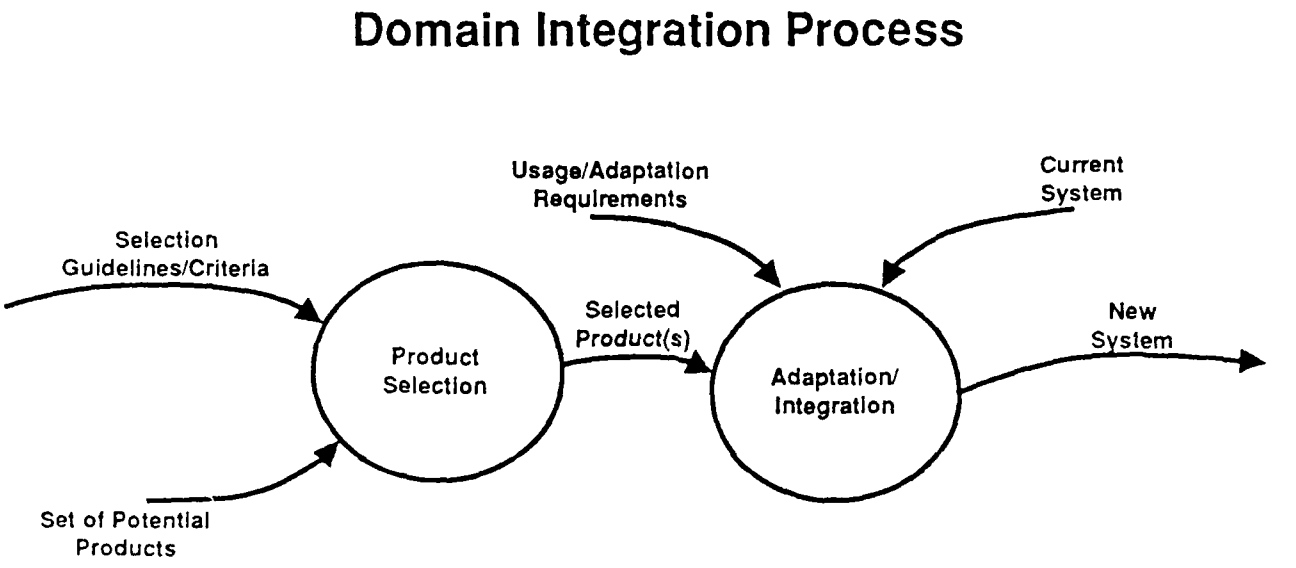


Figure C.15-2 Domain Integration Process

Table C.15-2 Competing/Cooperating Reuse Interest

Competing/Cooperating Reuse Interests

Domain Promoters	End System Promoters
Domain Developers	System Developers
"development for reuse experts"	"development by reuse experts"
supply reusable components	build application systems
System Refiners	Domain Integrators
"experts on system reuse"	"experts on domain reuse"
make application components reusable	adapt reusable components for application

The major problem NASA is having developing the third baseline is the mix of documentation and methods used to build the existing systems under analysis. They found they need to map all of the information to a common model and are trying to use an object-oriented model to accomplish this task.

The working group concluded that future system requirements must be included in a domain analysis to reduce the need for changes to the domain model. The group also concluded that change impact analysis is a critical domain analysis task. Many decisions are involved when changes are identified for a domain, with the most important consideration being existing users of the domain. If the change has minimal impact on users, the current domain definition could be modified. If the change is compatible with the existing domain definition, a new variant of the domain could be created. If the change is not compatible with the existing domain definition, then a new domain may need to be created. Figure C.15-3 illustrates some of the ways a change to a component of domain model could impact the model.

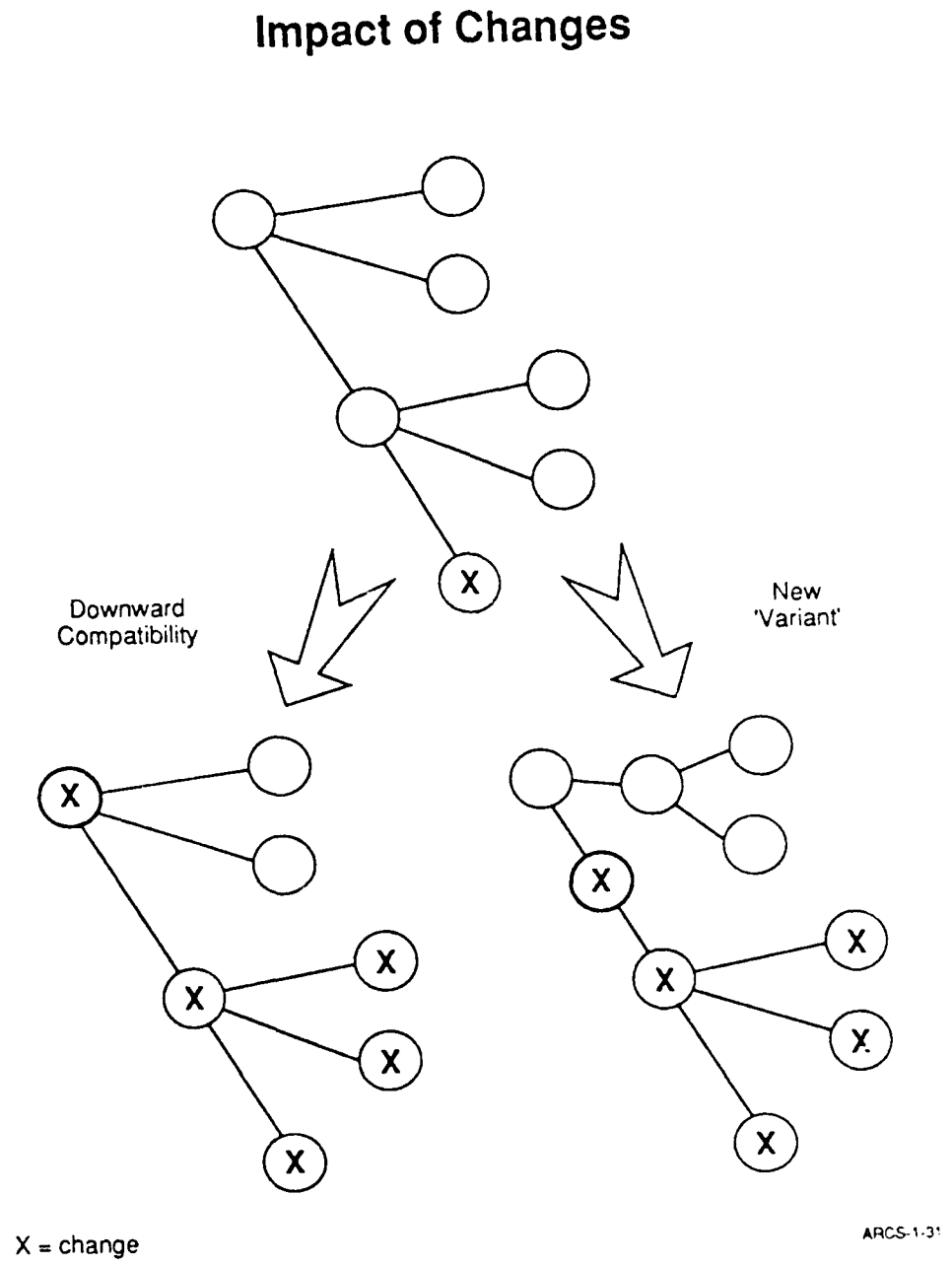


Figure C.15-3 Impact of Changes

C.16 Rockwell International

Rockwell's plan for developing reusable Ada software for the DoD and NASA includes the following steps [DIL85]:

1. Find general areas of commonality between software used in aerospace applications.

2. Determine functional areas within the general areas that can serve as a basis for reusable software.

3. For each functional area, determine the level that would best serve the purpose of a reusable package.

4. Implement the package according to various reusability techniques and standards for Ada packages.

A recent contribution from Rockwell's effort are concepts and guidelines for increasing the commonality of reusable Ada software. According to Nise and Giffen [NIS86], "commonality is two-dimensional. Software reuse can be measured by the degree of applicability of the package both within and across applications areas."

They propose the classification of software components according to a commonality matrix based on the expected amount of reuse along the two dimensions (see Figure C.16-1). A detailed domain analysis must be performed to identify the possible users within and across all domains. This requires analysis according to two mind-sets, application-oriented and functionally-oriented. The objective is to see any possibilities of expanding the domain of applicability of a component as early in the development process as possible.

"Maximizing commonality can be accomplished by thinking in terms of functions rather than applications areas and partitioning application-specific software from the functions that cut across many applications areas." A couple of techniques for accomplishing this for Ada packages are described. They warn against the potential negative effects of domain dependence, namely, the proliferation of packages within the reusable software library, and would not allow application-specific software to become part of the library.

WITHIN APPLICATIONS \ ACROSS APPLICATIONS	VERY NARROW	NARROW	WIDE	VERY WIDE
VERY NARROW	0	1	2	3
NARROW	1	2	3	4
WIDE	2	3	4	5
VERY WIDE	3	4	5	6

Figure C.16-1 A Commonality Matrix

C.17 Schlumberger Austin Systems Center and Schlumberger-Doll Research

Sayrs and Peltier [SAY88] discuss some of the problems in applying object-oriented approaches to domain analysis. In particular, they identify the choice of a taxonomic or non-taxonomic object organization as an "important dichotomy that exists in applying the object approach to one's domain." They believe the two approaches be "combined with advantages in efficiency and design clarity."

The authors define a taxonomic model as one in which "objects are related through the Class-Subclass hierarchy. Inheritance of data and procedures is through class ancestry." A non-taxonomic model is defined to be one in which "objects are related through domain-specific links. Inheritance of data and procedures is through path specifications."

They start by noting that "[m]ost object-oriented languages support taxonomic data types by providing 'built-in'" relationships and operations, with the application being responsible for managing links outside of the taxonomic hierarchy. "This is cause for concern in light of the often heard remark: 'real world problems are not inherently hierarchical, and therefore not well suited to taxonomic analysis." "We propose that the semantics of taxonomic inheritance, which vary among implementations, are not well suited for modeling components."

They found that a 'correct' taxonomic specification provides a good way to combine models to reuse 'overlapping parts', but the problem is that "it is not always possible to fit the taxonomic model to a problem domain." One problem with class hierarchies is projecting 'parts/part-of' relations onto instances (e.g., the actual components in a library).

"In an operational sense, non-taxonomic relations are one-to-many relations among instances ... Unlike taxonomic links, the semantics of non-taxonomic links depend on the object classes being linked. In addition, classes presumably have many non-taxonomic links associated with them ... It then becomes necessary to specify a path for inheritance." The authors proposed representing the links as objects. (Note: Although not stated, most object-oriented languages model links as direct pointers to other objects, not as objects having semantics for their management).

Non-taxonomic relations must be defined at the class level so that "they can be managed in a way that is compatible with the management of taxonomic relations ... We want to be able to specify both types of relations on a collection of objects and not have the inheritance mechanisms interfere with each other. Even more, we would like to be able to write generic functions that are indifferent to the operation of the inheritance mechanisms."

"Domain-Specific Automatic Programming" [BAR85] addresses several critical issues in acquiring and applying domain-specific knowledge in creating code, including impacts on the reusability of the knowledge. This paper was the result of the ONIX project at Schulemberger-Doll Research, whose domain of interest is oil well logging.

The author defines an automatic programming system as one where a *computationally naive* user can describe the domain problem and concepts in natural terms, while accepting the informality, imprecision and omission of details inherent in using such a language. The implication of his definition is that it requires the tool to be "domain-specific, both in order to interact effectively with the user and to draw upon domain-specific knowledge during the implementation process."

He argues against the sufficiency of a general purpose automatic programming system (which takes a completed formal specification of a program as input) because their specifications often fall short in supporting the specification of critical domain knowledge. "First, the required domain knowledge is much more diverse than just definitions - it ranges from problem-solving heuristics to expectations about the run-time characteristics of the data. Second, most of the domain knowledge is relevant for many different programs - coupling it too tightly to the specification of a single program restricts the reusability of the knowledge. Third, it would be difficult for a computationally naive user to express the domain knowledge unless the automatic programming system knew a significant amount already."

One solution is to collect domain-specific knowledge as part of the process of specifying every program. As more and more programs are specified, the system's knowledge of the domain grows substantially. The first users would have to be very sophisticated, but eventually, the goal of supporting computationally naive users would be met. While Barstow sees this as a good approach for the long-term, he feels that it is not feasible for the short-term "because tool little is known about how to organize and structure the domain knowledge and about how domain knowledge and programming knowledge interact. In fact these seem to me to be the key research issues in automatic programming at this time ... Addressing these issues requires experimentation in the context of specific domains."

Barstow describes the results of an experiment with two domains, logging information from well sensors, and interpretation of the logs. The logging process includes data acquisition and process control capabilities which "must perform ... within real-time constraints; thus, efficiency is a critical issue." The interpretation software does not have the same real-time constraints, but must make qualitative inferences from relatively little quantitative information. Both domains involve the manipulation of large amounts of quantitative data. While the size of the resulting programs is relatively small (a few hundred to a few thousand lines of code), they must be integrated into systems which are several hundred thousand lines of code. "These two domains are instances of fairly broad classes of software ... Thus, there is some hope that a framework for automatic programming that applies to these domains would generalize to much larger classes of software."

In order to identify the knowledge required by the system, they developed hypothetical syntheses of particular programs. "A hypothetical synthesis is a sequence of reasoning steps that an automatic programming system might follow in writing a program."

The hypothetical synthesis involves two major activities, **formalization** and **implementation**. Formalization "begins with an imprecise and incomplete specification" of the problem and ends with a formal specification "of precisely what the program should do but not how to do it." There are six steps in this activity:

1. Produce a natural language specification of the problem. "Such a description may involve many domain-specific concepts."

2. Convert the specification into a schema-like form (described below). "Such a description is informal in that it may be incomplete (e.g., not enough information to compute the outputs from the inputs) or not quite correct (e.g., some inputs may be missing, others unnecessary)."

3. Identify a solution strategy according to the kind of problem being analyzed. That is, find a relationship producing appropriate output for the possible inputs (a solution). The strategy must deal with incompleteness or inconsistencies.

4. If a simple formal relationship is known, then specify/use that relationship.

5. If the solution is too complex (there are too many possible relationships), break the problem down (decompose the specification into subspecifications) using heuristics. Both domain-specific and domain-independent heuristics might be required. At some point, the selection of a specific relationship "must be made on the basis of assumptions or preferences expressed by the user."

6. Steps c through e are repeated until all the relationships are identified for which the inputs produce outputs according to the problem constraints.

Implementation "describes, in domain-specific terms, the major purpose of the program and the assumptions that may be made when implementing it." It starts with the formal problem specification and ends with code in the target language. "In trying to determine such a program, knowledge of programming techniques and mathematics plays a central role, although knowledge of the domain is occasionally required." There are five steps in this activity:

1. For each component of the specification, perform steps b) through d). "The overall process is one of gradual refinement, ending when the program description involves only concepts that are available in the target language."

2. If the component is in an algorithmic form, then "refine the algorithm by using refinement rules, selecting among alternative rules on the basis of efficiency considerations." Refinement involves replacing abstract concepts by more concrete concepts. Both domain-specific and domain-independent rules may need to be defined and applied (examples include data structure mappings, approximation of bounds, etc). Such refinement rules may also depend on the target architecture.

3. If the component is not yet an algorithm, try to use mathematical rules or techniques to find a different but equivalent description, with the hope that it will be more amenable to direct translation into an algorithmic form (for example, an algebraic manipulation or other rearrangements). Barstow calls this *reformulation*.

4. Convert the component to algorithmic form (for example, convert postconditions into algorithms). If one algorithm if found, then use that one. If multiple algorithms are found, then select one based on an efficiency analysis or other heuristics.

5. At various times, "it may also be advantageous to apply optimizing transformations to the algorithm." This step might also apply to the combination of algorithms to support the next level of refinement.

Barstow identified "three qualitatively different types of descriptions" involved in the process: informal specifications, formal specifications, and algorithms.

Information specifications are "an imprecise, incomplete description of the program." Interestingly enough, this *does not* refer to the natural language specification, but the initial formal schema.

Formal specifications are the final result of the formalization process, which contains descriptions of inputs (including their data types), outputs (including their data types), assumptions (or preconditions), and postconditions. "The pre- and postconditions are formal statements, generally in mathematical terms, of the relationships that may be assumed about the inputs and that must be satisfied by the outputs."

Algorithm is "a procedural description of a technique for computing the outputs from the inputs." It consists of inputs, outputs, control structures, and imperative and applicative operations. "During the implementation activity, an algorithm may range from abstract to concrete: the formal specification is initially transformed into an abstract algorithm which is then refined into a concrete algorithm. The difference between the abstract and concrete levels lies in the data types and operators." (An example of an abstract data type is a "mapping", and an example of a concrete data type an "array." Intermediate levels are also possible).

"Although we may distinguish among these three types of program description, it seems important to have a notation that spans all of them." Such a language is called a *wide-spectrum language*. One reason for having a wide-spectrum language is to support development of different components at different rates. "For example, it may be wise to implement one component, or order to ensure that it is possible to do so, before attempting to work on another." The second reason given was to support the development of tools. "These are most easily built if there is a single coherent language upon which they can operate." They recommend connecting the components together based on their data flows. Any component in the network could be at any level of description.

Barstow identified four classes of knowledge that were required to accomplish the synthesis: **knowledge of programming, knowledge of application domain, knowledge of mathematics**, and **knowledge of the target architecture and language.**

1. **Knowledge of programming**. "The formalization process required domain-independent problem-solving heuristics ... The implementation involved knowledge of abstract data types and the associated refinement rules." Knowledge of specific programming techniques and algorithms, and of the efficiency of alternative techniques was also required.

"Programming knowledge would be represented in several different ways. Domain-independent problem-solving heuristics would be represented as pattern-action rules ... Knowledge about data types and operators would be represented in a hierarchically structured knowledge base ... Knowledge about implementing the data types and operators would be represented as refinement rules ... [a]dditional knowledge about programming would include algorithms to perform efficiency analysis or symbolic execution."

2. **Knowledge of application domain**, including basic facts about the subject matter, domain-specific problem-solving heuristics, and domain-specific data types and their associated refinement rules.

"Domain knowledge would also be represented in several ways. Basic facts and relationships would be represented as structured objects, and stored in a knowledge base. Domain-specific problem-solving heuristics would be represented in a fashion similar to the domain- independent problem-solving heuristics. Finally, knowledge of domain-specific data types would be represented as extensions to the system's knowledge of domain-independent data types and refinement rules."

3. **Knowledge of mathematics**, consisting primarily of techniques for manipulating mathematical expressions, which were applied to the synthesis process. "These would be represented primarily in a procedural fashion."

4. **Knowledge of the target architecture and language**, including number of processors, and language syntax. Knowledge about the target language "would be represented essentially as a filter on the allowability of refinements into data types and operators ... those that are inappropriate for a given target language would simply be tagged as such ... Knowledge about the syntax of the target language would be represented as a translator from the lower levels of the wide-spectrum language into the textual form required by the target language's compiler or interpreter. Finally, knowledge about the execution costs of the various operators in different languages would be stored in the operator hierarchy. Knowledge about the software architecture would be represented in a way similar to knowledge of the target language." That is, processes could also be tagged as inappropriate for a given architecture.

Barstow makes several observations about the syntheses activities. "As suggested by the above descriptions, there is actually a tree of program descriptions to be explored. The successors of a node in the tree are determined by the transformations that are applicable to that node. An interesting point of comparison between the two activities is that the trees associated with them are qualitatively different. The formalization tree is broad an shallow with many dead-ends; the implementation tree is narrow and deep with relatively few dead ends. Thus, the strategy for formalization is aimed at finding any successful path, while the strategy for implementation is aimed at eliminating as soon as possible paths that lead to inefficient implementations."

Barstow also addresses the evolution of domains. Knowledge of " ... our application domains, as well as most other domains ... evolves over time. One consequence of the separation of the programming process into the formalization and implementation activities is that different types of changes in the knowledge lead to changes in different activities." Changes in domain facts (e.g., the best equation to solve a certain problem) would impact both the formalization and implementation. A change in target architecture (e.g., adding more processors), however, would only affect the implementation. "In both of these cases, the framework described earlier lends itself to recording the various decisions that were made initially during either activity, which would in turn support the selective re-derivation of a program to reflect either type of change." Barstow concludes with some discussion of current issues in automatic programming. A few interesting points were made:

- In mathematical domains, algorithm design and selection requires some "creative insight or discovery, some kind of 'Eureka!' In our domains, the major difficulty is simply the large amount of domain knowledge that must be understood and applied."

- o Interactions between components can be a problem, but did not come up as a major issue for their domains. Reasons were the high cohesiveness of the algorithms, and standard representations of the data to be passed between components.
- o Constraint propagation is important to their domain, "especially for dealing with exceptional conditions."

- o Current general-purpose generators often embody domain and programming knowledge directly in the language and algorithms. Such an approach cannot be used for their domain because components whose specifications are similar could be implemented in vastly different ways due to mathematical, domain, and target considerations. Also, their domains require qualitative reasoning, where the model may change based on domain-specific assumptions.

- o The key distinction of their domains over others "seems to be that the complexity is due to the sheer volume of domain knowledge that is involved, rather than to the domain-independent algorithms and data structure choices that the software embodies. A second important characteristic is that much of that knowledge seems to be important for many different programs within the domain. These both seem to be characteristics of many real-world domains ..."

Barstow compares Schulemberger-Doll Research's approach with DRACO and PSI. Both their approach and DRACO emphasize the use and reuse of domain knowledge. DRACO supports their "belief that realistic real-world domains require substantial amounts of domain knowledge; for example, the domain model for one rather specific domain (tactical display systems) was over 100 pages long. In terms of the distinction between the formalization and implementation activities, DRACO seems to be focused primarily on the second."

"PSI's distinction between acquisition and synthesis was similar to our distinction between formalization and implementation ... PSI also included a 'domain expert' in its initial design, but this was eventually absorbed into the Program MOdel Builder, one of the components of PSI's acquisition phase. Thus, PSI placed considerable less emphasis on domain knowledge."

C.18 Software Engineering Institute

According to the Software Engineering Institute (SEI), a domain analysis is performed as the first step of the following six-step process for reuse (see Figure C.18-1, SEI Six-Step Process for Reuse) [SEI88a]:

1. **Construct** the reuse methods and products
2. **Collect** the reuse products into a library
3. **Catalog** descriptive information about the products
4. **Classify** the products according to semantic/functional information
5. **Consider/Comprehend** the available resources
6. **Customize/Compose** the application from reusable resources

The reusable products, also called reusable resources, are divided into several categories (requirements, parts/code, etc). The SEI addresses both feeding the domain analysis results forward into parts construction, and feeding back any additions or revisions to the reusable resources during parts construction.

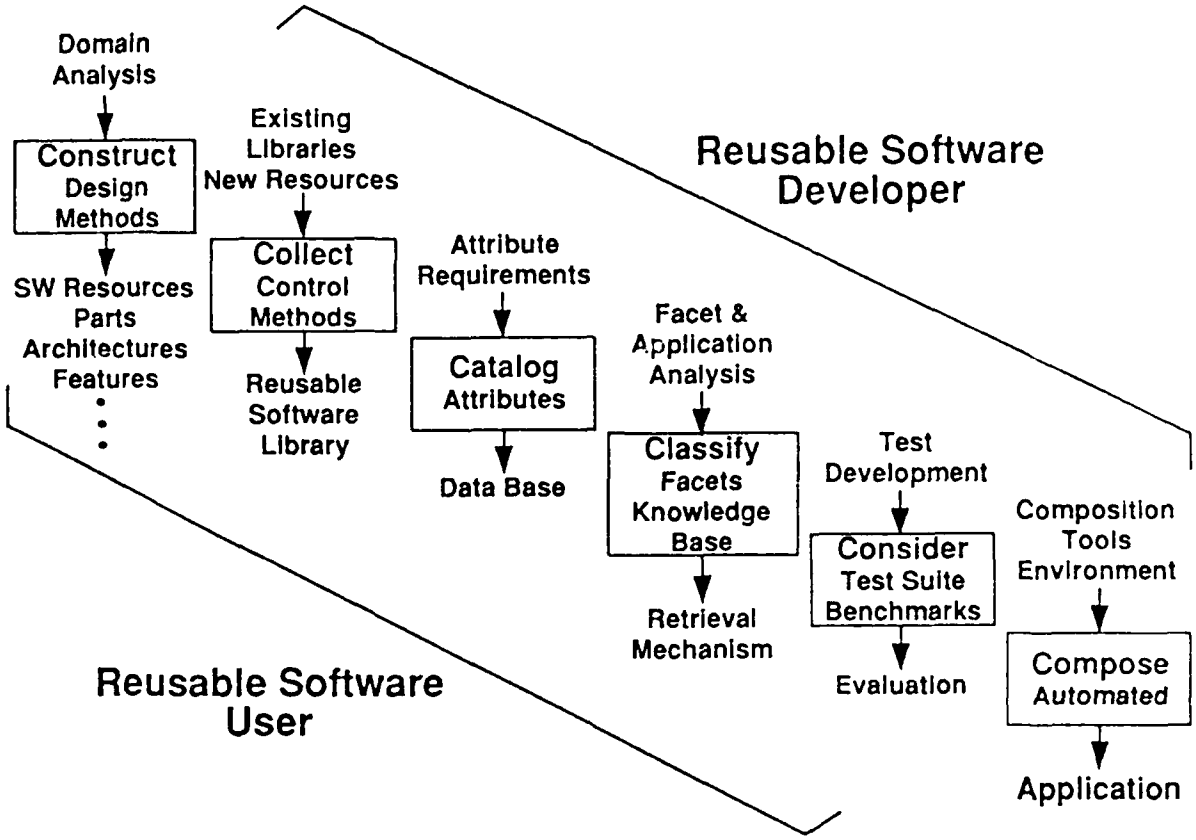
Domain analysis is broken down into two distinct analyses performed roughly in parallel: the **features analysis** and the **commonality study**. **Features analysis** is a requirement: -based analysis of the domain. The domains/subdomains are identified during this analysis, as well as candidates for the commonality study. The products of this activity are a domain taxonomy, domain feature descriptions, and other support for reusable resource identification.

The **commonality study** is an architecture- or design-based analysis of the domain. The selected candidates are analyzed to identify features which are closely related or features which are shared by two or more candidates. Requirements for the next level of analysis are defined.

The domain analysis process is applied throughout the development process, with the specific analysis activities differing based on the level of development as discussed below:

- **Systems Domain Analysis** - The features analysis at this level looks at system-level requirements, and selects the actual systems to serve as candidate applications. The commonality study looks at how the system-level requirements have affected the candidate software, and explores features found missing in one or more candidate applications. Both functional and environment requirements are addressed, with a data flow orientation applied.

- **Software Requirements Domain Analysis** - At this level, requirements are generated for the features identified by the system-level commonality study. Relationships among domains are established, including considerations of domain dependence and independence. The outputs at this analysis level are specifications for the reusable software parts.



Developer/User View of Process

Figure C.18-1 SEI Developer/User View of Process

- **Design Domain Analysis** - An algorithmic analysis of the specified parts is performed at this level, addressing factors such as timing and sizing, accuracy, and computational methods. An approach for constructing the parts is also identified at this time. The parts are designed as Ada packages or subprogram specifications, with generics used at several levels. Detailed design implements the bodies of the parts. Test code is developed at the unit level for subprograms and integration level for packages, and is considered a reusable resource.

This effort identified several organizational approaches for developing reusable software, and looked at the impacts. With respect to domain analysis, several impacts were extracted from their findings:

- A project spin-off effort for developing reusable resources does not perform a domain analysis, and so the results are not likely usable for other projects.

- An autonomous parts group begins with a domain analysis in generating reusable resources. The results are more likely to support multiple projects, but by operating in a vacuum, may end up failing to support the needs of specific projects.

- A project-directed parts group works with specific projects during domain analysis and throughout the development of the reusable resources. The resulting products are very likely to support multiple projects as well as meet the needs of specific projects.

The methodology was applied to a reuse experiment, a redevelopment of the CAMP missile domain parts.

C.19 SofTech

The SofTech *Ada Reusability Guidelines* [BRA85] and Goodenough's early work on the effect of software structure on reusability [GOO74] propose a software development approach that could be applied to domain analysis. The recommended approach is to derive a generalized architecture model for an application. The model provides a common viewpoint for stating software needs and matching these needs against existing capabilities. Designing software to fit the model increases the chance that it will be able to be reused in other systems.

SofTech further recommends that the architecture model should be layered. A layered architecture enhances reusability in that it lets the reuser replace individual layers independently when tailoring the function or performance of the system. Thus change effects are confined to a relatively small part of the system. [BRA85]

SofTech developed an approach for applying generic architectures in more detail in [QUA88]. The remainder of this section summarizes the results.

"Generic architectures provide an approach to the development of reusable software for families of related applications. They provide both a high level design and a set of reusable components to be used in the applications supported by the architecture. The components are typically larger and more complex and result in higher levels of software reuse than with conventional reusable components. The study explores the use of the Ada language in the development of generic architectures for Army command and control applications ... "

"In many respects the implementation of a generic architecture is like that of other software. However, there are three areas in which special attention is required. These are in the definition of the application domain, the design of the architecture, and the implementation of the components."

The author provided little information about defining the application domain. "There are no hard and fast rules that govern the definition of a domain, but rather several things that should be considered."

- o The applications must be similar enough to reasonably share a common design and set of components.
- o Commonality of hardware and software environment makes it easier to develop reusable components.
- o There must be enough applications to justify the cost. "Enough" varies with degree of variation in the applications. For very similar applications, only a few are needed. For domains with greater differences, a larger number are needed to offset the additional cost to make them more adaptable.
- o The domain should be within the scope of a single organization or project office. This organization provides funding, monitors the development, resolves issues, supports its use on applications, and receives the benefits of reuse.

The purpose of a domain analysis is to identify the common requirements of the applications covered by the domain. "For a generic architecture, it should cover not only the applications, but also the hardware and software environment. For software with a potentially long life cycle, it is

important to understand the types of changes in hardware, software, technology, and applications that it might have to accommodate in the future."

Several guidelines for the process are provided. Begin by listing difficult design decisions or design decisions which are likely to change; these should be hidden in the design to minimize the impact of changes. The designer must have sufficient experience and ability with the domain covered by the architecture. Designing a single complex application is difficult; a generic architecture presents many additional requirements.

Two approaches are considered. One involves developing the architecture on the basis of the previously-implemented applications. The second involves developing a prototype of the architecture under consideration.

In the first case, the existing applications serve as prototypes for the generic architecture. In any given domain, it isn't unusual to find some existing application(s) that is somewhat representative of the domain as a whole. The design of the generic architecture should not necessarily be the same as the existing applications; however, this depends on the generic design considerations mentioned earlier.

In the early stages of prototype development, it is best to concentrate on developing early versions of the *principal* components, omitting many of the minor ones. Develop two or three actual applications; analyze the results to identify changes required for the production version of the architecture. This analysis will also identify additional components to be made reusable.

Use the "principles of object-oriented design" to define the objects around which the reusable components will be designed; each object must be described by a set of variables and a set of operations that can be performed on instances of the object. "The packages representing these objects are the reusable components of the architecture ... The components are highly integrated and intended to be used together ... Components that will be different for each application are often represented by 'dummy' versions which simply serve as place holders ... Additional components, not provided with the architecture, are required for most applications."

The components of a generic architecture are likely to be larger and more complex than a library's components. It is also important to remember that "the components are not intended to be reusable outside the application domain of the architecture."

SofTech provides several guidelines for designing the Ada components, including use of generics, etc. They identified several unanswered questions whose answers "require actual experience in the development of generic architectures in Ada":

- o How serious are Ada's limitations in supporting inheritance?
- o How well can dependencies be isolated?
- o What are the limitations on types of applications that can be supported by generic architectures?

The report then describes a strawman generic architecture for the tactical portion of Army command and control applications, specifically based on ACCS. There are some important facts to note about this domain:

- o The functional segments are in various stages of development, and are expected to be maintained and enhanced well into the 21st century.

- o They are expected to be maintained and enhanced well into the 21st century
- o Numerous changes can be anticipated over its life cycle, "both in Army doctrine, with its impact on force structure, information flows, and processing responsibilities, and in the technology of computer hardware, software, and command and control."
- o Support for distributed operations, implying support for redundancy and parallel processing.
- o The applications use (or will use) common hardware and software.
- o Program development will occur in an evolutionary fashion program with meaningful baselines released during the development.
- o Man-machine interface requirements are often implied.

In addition, although there are many requirements unique to each segment, some are shared across all segments. SofTech looked specifically at the maneuver control segment. Security, training, performance, flexibility, reliability, interoperability, maintainability, portability, and efficiency requirements augment the functional requirements; these are likely to be shared across segments.

SofTech found few internal and external dependencies that would preclude reuse. However, hardware and software environment dependencies are the most risky kind. The Army must promote industry standards and isolation of such dependencies to protect its investment in ACCS software.

The functional requirements of the maneuver control segment address general capabilities. The application details are addressed by a data-driven approach; because of this, they are likely to apply to other segments as well. However, other segments address much more specialized missions.

The report describes the work that would be required to demonstrate and validate the generic architecture approach. The effort would have two objectives: to demonstrate and validate the generic architecture approach, and to develop a prototype generic architecture along with a set of applications that use it on a specific project.

To achieve the first objective, it is necessary to reveal the details of the methodology at each step in the process, to evaluate and review the process regularly, and to document the lessons learned. The second objective requires achievement of the first objective, and also provides the vehicle to support it.

"A generic architecture is inherently project-specific." Therefore its development must be sponsored by the organization responsible for the project. "No one else can provide adequate access to information on the requirements of the applications or evaluate the usefulness of the results ... Execution of the process without the active participation of such a sponsor would seriously compromise the usefulness of the experience in the future development of generic architectures on other projects."

The three steps in the process are:

1) Perform a domain analysis.

2) Develop a prototype generic architecture.

3) Develop prototype applications using the generic architecture.

"A domain analysis is conducted to establish the scope of the application domain supported by a generic architecture ... the domain analysis includes the effort normally associated with requirements analysis and preliminary design. Because the applications within the scope of the domain must share a common design and components, the top level design is needed to determine which applications are within the scope of the domain."

SofTech recommends using the domain analysis effort to provide the vehicle to *develop*, *demonstrate* and *analyze* the methodology to be used. Basically, this says they do not have a methodology, so they will develop one "on the fly" as part of the prototype architecture development.

They note that the *problem domain* (the domain covered by the analysis) and the *architectural domain* are "not necessarily the same." These are the distinctions they make:

- o The problem domain includes all applications of interest to the sponsor; the architectural domain only includes all applications that can share a common design and a significant number of common components.

- o A problem domain may include several architectural domains. "A group of applications within the problem domain may have enough similarities within the group to meet the requirements of an architectural domain, but have enough differences from other groups to prevent the sharing of a single architecture among the groups. In this case a separate architectural domain would exist for each group of applications."

- o Not all architectural domains are important enough to justify the development of a generic architecture.

The same approach is applied to the development of multiple generic architectures for a problem domain if that were necessary.

"A domain analysis should include the following five major activities:

1. The definition of the problem domain,

2. The identification of the software requirements for the applications within the problem domain,

3. Analysis of those software requirements to determine the appropriate approach to software reuse,

4. Development of the requirements and top level design for the reusable components, and

5. A cost benefit analysis for the reusable components."

"The results of the domain analysis should be documented for review by the sponsor and others interested in the program. The documents should include:

- o The System/Segment Specification (SSS) for the applications in the problem domain,

- o Software Requirements Specifications (SRS's) for each application in the problem domain,

o An SRS and a Software Top Level Design Document (STLDD) for the reusable components that will be shared by the applications,

o A cost benefit analysis, and

o An analysis of the lessons learned in the domain analysis."

Additional details were then provided for each of the activities identified above.

Defining the Problem Domain:

Include all of the applications of concern to the sponsor. Reduce the size of the problem domain by eliminating applications too small or temporary to benefit from the reuse of components. Describe the domain in a SSS, where each application is identified as a separate CSCI; "An additional, separate CSCI should be identified for the reusable components that will be shared by the applications."

Identification of Application Software Requirements:

Develop an SRS for each CSCI, organizing the requirements to meet two objectives: support the selected approach to software reuse, and identify common functions that might be supported by reusable components. Those objectives are met by using an object-oriented organization of the functional requirements; each of the major functions would be "approximately equivalent" to a component in the generic architecture.

"[T]he description of those major functions should be as similar as possible across the CSCI's so as to highlight both the similarities and the differences that must be recognized in the design of reusable components for those functions." They must also be in a "style that will allow the sponsor to confirm" completeness, correctness and adequacy of this restatement of the original requirements.

An object-oriented approach has the following advantages:

o The components tend to be "inherently reusable"

o Large reusable components are already likely to be object-oriented

o OOD highlights opportunities for reuse

o OOD facilitates traceability of design to requirements

Determination of the Approach to Software Reuse:

Review the requirements to determine which requirements are common to a large share of the applications, and which applications are likely to share the initial top level design. "The selection of the generic architecture approach to software reuse is appropriate if:

1. The top level design can be shared by an adequate number of applications, and

2. Those applications have a significant number of similar functions."

A library of reusable components not associated with the generic architecture would be "justified if the applications contained a number of functions that were common to two or more applications and that were relatively independent of the design of those applications."

In addition to the criteria presented above, an analysis of the costs and benefits must be performed to justify the approach. Even if the generic architecture approach is selected, a separate CSCI, with its own SRS, should be created for the reusable components to be developed using the library approach (whose components will not share the top level design associated with the components of the generic architecture).

Development of the Reusable Component Requirements and the Top Level Design:

Develop Software Requirements Specification(s) for the reusable components; each component should be treated as a separate major function. Identify dependencies that should be isolated to facilitate later changes to the architecture and its applications. Identify parts of the design that may have to be adapted to the requirements of individual applications and the types of changes that may be required.

Complete a STLDD for the CSCI which covers the reusable components of the generic architecture; the STLDD will contain:

- o Architecture
- o Control relationships
- o Data global to the architecture or component
- o Details of TLCSC requirements
- o Ada package specifications for each component
- o Adaptation requirements for each component
- o Assessment of the likelihood of requiring adaptation

Resist the temptation to build in generality that is not required by the requirements of the architectural domain. "The efficient use of this approach depends on the ability to reduce complexity by tailoring the architecture to the specific requirement of its domain."

Cost Benefit Analysis:

The generic architecture approach "can result in a significant reduction in the life cycle cost of those applications" which will make use of it. However, the initial development of a generic architecture is more expensive than that of an actual application of similar size and complexity "due to the additional analysis and design effort that is required ... The cost of this additional effort must be incurred before the architecture can be used to support its first application." This analysis should precede a decision to implement the architecture.

The task is made easier because the domain is bounded. "It is possible to make reasonable estimates of the number of applications that would benefit and the contribution that would be made to each application." The benefits "are likely to be both quantitative and qualitative."

"Available costing models can be quite useful in estimating the impact of using a generic architecture on the life cycle cost of the applications within the architectural domain." First estimate the life cycle costs both with and without an available generic architecture. The generic architecture estimate "would be based on the number of lines of code that would have to be added or changed and the complexity of the task of extending the existing design provided by the architecture. It would also require an estimate of the development cost of the architecture itself,

based on estimating formulae for software of similar size and complexity. Allowance would have to be made for the additional effort required to carry out the requirements analysis, design, and testing of the architecture."

Identify qualitative impacts "explicitly for the benefit of the decision maker." Such impacts include these:

- o Situations where interoperability will be improved
- o The isolations of dependencies that will facilitate use of the architecture in other environments
- o Potential lessening of accountability of those responsible for implementing the applications; problems may be more easily described as being "out of their control" unless they are also responsible for the generic architecture.

Several notable points were extracted from their discussion of developing the prototype generic architecture:

The prototype only supports "the principal functions of the final product", and so does not account for the special situations that "can account for a substantial share of the total development cost." It is only "a logical first step in the evolutionary development of a software system."

A prototype generic architecture "may not do anything"; prototype application code which surrounds it is needed to test the architecture.

The development of the prototype application begins with the results of the domain analysis, followed by detailed design, coding and unit testing, CSC integration and testing, and CSCI testing.

The top level design produced during the domain analysis will be more complete than for a "normal" development because more detail is required to determine if the architecture is adequate for use on a given application.

The SDDD contains the detailed design for the operations of the reusable object-oriented components, and includes adaptation instructions.

"Special attention should be given to the planning of unit tests for the components to reduce the need for scaffolding to represent components that are not yet available." Additional code will be needed to test the adaptation mechanisms.

"CSC integration and testing covers informal testing on aggregates of reusable components. It should be supported by a build plan for the components that allows the architecture to be used in the development of primitive applications as soon as enough components have been completed to support those applications."

"CSCI testing is the formal testing of the generic architecture, and it should be performed by an independent test team. The tests themselves should be based on documentation contained in the STLDD and SDDD on the role and adaptation of each component. Final CSCI testing will be performed with the prototype applications."

Points were extracted from their discussion of developing the prototype applications:

The prototype provides a test of the reusability and adaptability of the design an components, in particular the adaptation documentation. It also *helps in understanding the technology* and in *understanding the contribution made by the generic architecture*. They recommended only prototype applications which were *subsets of those identified for the application in the original domain analysis*; the only application-specific requirements allowed were those that test the adaptability of the architecture or the ability of the design to support the application.

Use of a generic architecture should streamline the remaining life cycle phases:

- o Preliminary design is only applied for significant extensions to the architecture, for new components, and for replacements to components.
- o Detailed design and coding is limited to and required for the items above, plus any significant adaptations of the reusable components.
- o The generic architecture provides the scaffolding and most of the drivers supporting the testing process; the generic architecture dictates the application build plan.
- o The independent test team must still work from the original SRS's for formal CSCI testing

In summary, Quanrud makes the following important points:

- o The technology is still experimental
- o SofTech's expertise is not easily transferred to other organizations.
- o The information used in the analysis was "quite limited."
- o The sponsor's participation in the demonstration is critical; selection of the "right" sponsor is critical.

Several issues were not covered by the study:

- o The number of "significantly different" designs of importance in the development of generic architectures.
- o The degree to which the design and components can be used for applications outside the domain.
- o The degree to which reusable components can be shared among generic architectures.

C.20 Software Productivity Consortium

Pyster and Barnes summarize the efforts of the Software Productivity Consortium in the area of software reuse in [PYS88]. Domain analysis is briefly described as one of the areas currently being addressed.

The authors note that there are "two methods for finding modules with which to stock a reuse library's shelves. The first is essentially ad hoc as targets of opportunity are presented ... Through proper guidance or luck he produces a version that is flexible enough for others to use. It is certified, cataloged, and inserted into the library. The strength of ad hoc collection is that it takes advantage of activities people are doing anyhow and requires relatively little advance planning. The weakness, of course, is that many opportunities will be missed. The second method is by performing a *domain analysis*; i.e., determining for a particular area such as flight navigation or database management which module families are important and describing both the families and their relationships. Such an analysis then becomes the basis for stating which implementations are needed to support that domain. Implementations can be sought from internal development or outside vendors depending on availability."

"The Consortium is mapping the CAMP work into its own framework, and attempting to generalize on it as a basis for conducting domain analysis in other areas."

The Consortium framework includes the notions of modules, families and reuse libraries. "The primary units of reuse are modules, and reuse libraries are organized around module families." A description of the family "states the basic properties and operations" of the family. "Each implementation, while unique, would still satisfy those basic properties an support those fundamental operations. A library user could read the family and module descriptions to understand the commonalities and variations among stack modules."

"There are at least two interesting relationships among module families. First, a family may be subdivided into smaller constituent families to facilitate explanation and organization ... Second, a family may be described using other families ... Major economies of description are possible by describing one module family in terms of another ... The description of one module family in terms of other family descriptions does not say anything about how the members of the first family are actually designed."

"A family description states what is common among the members of the family and what differences among members are anticipated ... It is important to note that complete behavioral equivalence among family members is not expected, or even desirable. Equivalence modulo the family description is all that is normally expected. This allows grouping of implementations that are *slightly* different in their behavior under a common name, reflecting how people would likely want to view them." The authors encourage supplying multiple implementations of candidate parts. "Although offering one implementation choice per family supports some reuse, it is usually too limiting for widespread application."

Another area of research related to domain analysis is the Synthesis Project, "chartered with constructing canonical designs for applications areas of interest, developing methods for constructing canonical designs, and where possible with constructing synthesizers around existing canonical designs." The synthesizer would use "a knowledge-based system to organize the domain analysis and select the correct members of each family in the one canonical design that has been build. In many cases the alternative implementations in a module family are actually constructed as one parametrized program segment, often as an Ada generic. The synthesizer determines the correct parameter values based on user input."

C.21 STARS

At the Applications Workshop for the Software Technology for Adaptable, Reliable Systems (STARS) Program, the participants of the subgroup on "Making and Using Reusable Parts" outlined a guidelines document for "Parts Requirements Analysis and Specification", containing the following information [STA85]:

- o Domain Analysis - "the process of analyzing *domains* in order to identify potential reusable parts."
- o Domain Identification - "refining the scope of the domain analysis."
- o Domain Representation - "choosing a set of applications (i.e., particular systems) to represent the domain of interest."
- o Commonality Study - "the study of the application set or order to (a) find common *objects, operations, and structures*, and (b) specify their requirements. An essential aspect of this study is evaluating the cost- effectiveness of the potential part."

The "Parts Taxonomy and Requirements" working g.oup made the following comment about domains: "A more complete definition and understanding of domain is needed. Domain affects the selection and composition of reusable parts but the degree and bounds of the domains effects are not known."

This working group proposed the following "levels of reuse":

- o Physical - "an implementation"
- o Logical - "ranges from 'designs' to 'algorithms' ... appears to be a range of abstraction"
- o Conceptual - "a range of abstraction that includes 'specification' and 'design'"
- o Environment - "a higher level of abstraction ... what it is or how it may be used is uncertain"

"At each level of reuse the lowest reusable component is a *part* ... At each level, the reusable parts may be incorporated into a new system in a number of ways, called compositions by the working group."

With respect to reusability support tools and aids, this working group notes that "[r]eusability will be futile if there are no automated support tools and aids. Development of these tools should be done on two parallel paths. Conventional technology, such as database management systems, will provide the most near- and mid-term support for selection, acquisition and insertion of reusable parts in a database ... The second path uses artificial intelligence and knowledge-based systems. There are some immediate applications that show promise, this technology will not provide the needed support for at least a decade, probably longer ... At the current level of technology, AI/knowledge based tools and aids are high risk and offer little immediate payoff."

The Libraries subgroup addressed a few issues associated with libraries and application domains. The group resolved that "[d]omain specific libraries are not recommended since there is significant overlap among domains. This would lead to confusion over which library a user should search. A single, centralized facility is, therefore, recommended."

The Design/Integration subgroup report indicates recognized the need for performing domain analyses for STARS. They recommended that two projects be undertaken in parallel: Horizontal

Domain Identification and Parts Development, and Vertical Domain Identification and Parts Development.

The objective of the Horizontal Domain Identification effort was "to identify domains within DoD mission-critical applications which span multiple application areas ..." The object of the Horizontal Domain Parts Development effort was "to identify, specify, construct and test parts from the horizontal domains. Award multiple contracts to demonstrate various parts development methodologies." The Vertical Domain efforts were similar, except that vertical domains are "critical application families ... in which there exists a high degree of reusability, a significant life cycle cost savings, and an opportunity for critical expertise propagation."

Another project was a "Software Reusability Demonstration," in which a mission-critical system would be developed "from an identified vertical domain using parts from horizontal and vertical domain preparation." Multiple awards were also recommended here to demonstrate alternative methodologies.

C.22 SIGAda

Members of SIGAda have recently formed a Reuse Working Group. This working group contains a subgroup which plans to address domain analysis and reuse libraries.

C.23 Unisys

"[T]he Unisys Reusability Library Framework (RLF) STARS Foundations project [UNI88] is intended to provide a general framework and a base set of tools supporting the creation and maintenance of repositories of reusable Ada software components, organized around particular application domains. The project is being designed and implemented by a team of 7 software engineers at the Paoli Research Center."

"Unisys believes that the most dramatic productivity gains in reuse will be realized through the development of libraries or components for specific domains, and structured according to explicit **domain models**. Such models can be realized as detailed taxonomies of the objects and operations which are pertinent to the different semantic attributes for cataloging and retrieving components, and the taxonomy used in each domain will evolve with usage over time. Thus, it is important to realize that there is no single "right" classification scheme for software reuse; what is needed are model-building tools which can be used in creating libraries for many different domains." The team also believes that it is important to support " mixture of constructive and generative approaches to reuse."

The team believes that domain models will contain both highly structured taxonomy knowledge and more loosely structured heuristic knowledge. For this reason, they are integrating a structured inheritance (semantic) network system with a production rule-based system. "... [W]e believe this hybrid approach will provide the base functionality necessary for search by keyword or arbitrary attributes, or faceted classification schemes." They have defined static specification languages for describing knowledge bases or domain models.s They are also "implementing translators to map these description data structures."

The Unisys team is also providing "a vase set of components for library tools that can be shared across libraries with very different domain models. The semantic network knowledge representation system (differing in this respect from object-oriented approaches) allows for a separation of declarative and procedural knowledge: that is, applications that traverse over semantic networks can be written to be relatively independent of the specifics of the model being traversed ..." In Entity-Relationship (ER) terms, this is a many-to-many relationship: a single application can access many models and many applications can access a single model. It is possible for someone to implement a collection of library tools that can be reused from one library to another. The team's library tools for component testing and library browsing are built on their Universal Browser, "a toolkit of general routines to access and manipulate model descriptions."

The team is also working on a **domain analysis process model** that will cover what they call the "domain life cycle ... [T]he process model should be used as a framework for identifying where tools (active agents) and instruments (passive collectors of information/knowledge) would fit into the life cycle." [SIM88]

C.24 University of California - Irvine

Arango's work at UC Irvine addresses what he calls "domain engineering" [ARA87][ARA88]. Domain engineering (DE) includes identifying, generating, representing, analyzing and evolving reusable resources. "We emphasize the term *domain* because application - or domain - specific knowledge is the dominant factor in the organization of reusable resources. We emphasize the term *engineering* because of the need for systematic, cost-effective approaches to find solutions to the infrastructure problems."

Arango sees two classes of problems involved in making reuse practical: operational problems (the mechanization of reuse) and infrastructure problems (the systematic, economic development of reusable resources). These two problems are interdependent as mechanizing reuse presupposes the existence of appropriate reusable resources, and generating those resources presupposes knowledge of how they will be reused. "That is, the generation of reusable resources is predicated on a model of the performance task to be supported." Arango believes that the infrastructure problem suffers from "a lack of theoretical foundation and of methodological support."

In a reuse-based task, there is a gap between the *perception* of similarities between a class of software requirements and the *generation* of reusable encapsulations to satisfy the requirements. Arango calls this the *problems/reusable encapsulations gap*. He compares bridging this gap to bridging a needs-program gap in software development: "Given a class of similar problems, how do we identify reusable abstractions to support the task of the reuser?" can be compared to "Given some set of requirements, how can we design and implement a program that satisfies them?"

"SE [software engineering] has developed systems of principles, methods, representations and tools to help bridge the requirements-program gap in a controlled, systematic manner." Arango argues for the same effort in domain engineering to bridge the problems-encapsulations gap.

The first problem is defining the term "domain." Domain can refer to a class of problems, a class of software systems or a collection of encapsulations of application-specific knowledge (e.g. a library of Ada packages for radar signal processing). This is due in part to the varying viewpoints of members of the reuse community.

"Members in the reuse community focus their efforts on different segments of the software development process - a *performance* task. These tasks are quite varied, for example, analysis of specification; formal specification and transformational implementation, software construction; maintenance; program synthesis; verification. The nature of the reuse-based task biases the viewpoint of the reuser on how to define what constitutes a problem, what are reusable problem or solution abstractions, and how to assess competence or performance in the reuse task."

Arango addresses the purpose of domain engineering both as a *conceptual framework* and its realization as a *methodology*.

Arango's team studied the infrastructure problem in a variety of reuse scenarios, and discovered four needs:

1. To define concepts that are becoming important to the reuse community

2. To find covering abstractions that allow both description and analysis of infrastructural problems independent of the reuser viewpoints

3. To identify requirements that could guide the development of bridging or DE methodologies

4. To develop a framework for assessing the worth of methods and representations.

Figure C.24-1 illustrates that "DE is concerned with supporting the generation of the infrastructure for reuse, and **not** the actual reuse process."

"The purpose of a DE methodology is to aid in the 1) identification, 2) acquisition, 3) justification, 4) analysis, and 5) evolution of reusable abstractions." Figure C.24-2 provides Arango's high level view of domain engineering, and the role of a DE methodology.

Arango believes that there are at least three kinds of reusable abstractions: abstractions for problems, abstractions for problem-solving mechanisms (task abstractions), and abstractions for solutions.

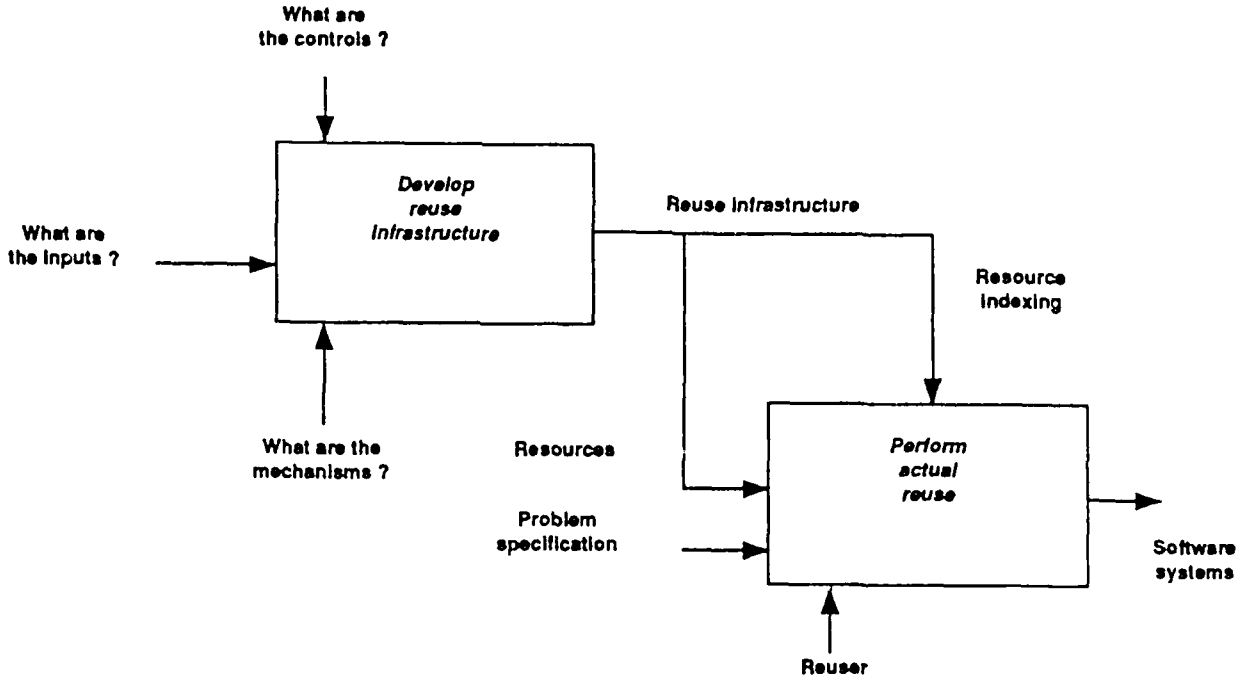
Arango points out the difference between reusable *abstractions* and reusable *encapsulations*. For example, "Sort List" is an abstraction, but can be encapsulated as an Ada generic or a Lisp function as dictated by the reuser's implementation. "... if the mechanical reuser is the Draco system, problem abstractions are encapsulated as constructs in formal languages, while program abstractions are encapsulated as tree-to-tree transformations. If the reuser is an Ada programmer, problem abstractions may be encapsulated (or encoded) as sets of keywords which act as surrogates for the problem semantics ... Program abstractions may be encapsulated as Ada generics."

Arango finds a "striking similarity" between manipulating reusable encapsulations and manipulating code in a conventional maintenance situation. "The same diseases are present: difficulties in understanding and managing the propagation of change, hidden assumptions or coupling between the semantics of different encapsulations, reusability attributes spread among several encapsulations to compensate for inadequacies in the encapsulation language, and so on." The analysis and evolution of an abstraction's reusability properties must be done at the proper level, and the encapsulation level is not the proper level. In [ARA], Arango states that working with the abstractions produced as a result of a domain analysis can greatly improve the performance of the design recovery process in maintaining and porting software.

Arango states that the study of the design and use of structures for encapsulating application-specific information for software construction is an important field of software engineering. But while *encapsulation engineering* "is a key participant in the technology assembly for reuse ... the problems of how to develop, analyze and evolve reusable abstractions define distinct domain engineering concerns." Ideally, encapsulating an abstraction is a semantics-preserving transformation. However, it is often the case that the encapsulation language cannot express certain semantic information (e.g., timing information), or the abstraction's reuse properties are adversely affected (e.g., polymorphism).

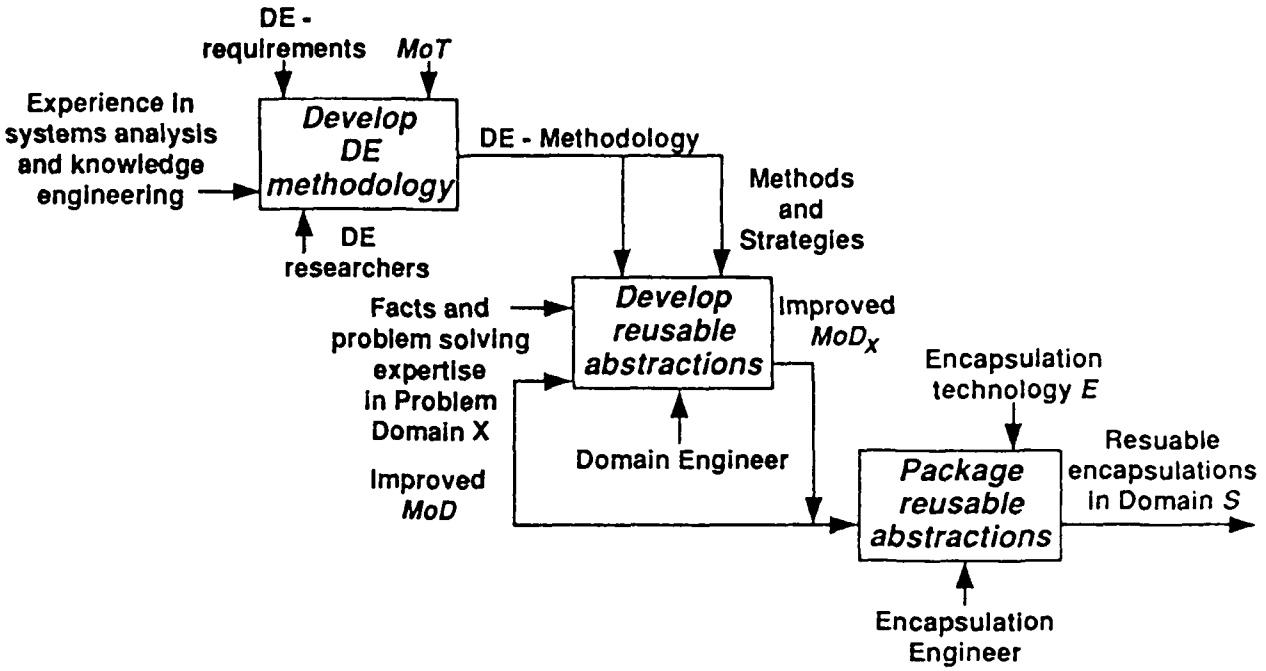
The following definitions highlight the fundamental concepts in Arango's view of domain engineering:

> **Model of the Reuse Task (MoT)** - a deductive model such that "there are a certain number of initial objects and a certain number of rules for generating new objects from the initial objects and from those already constructed."



First Develop an Infrastructure, then use it

Figure C.24-1 First Develop an Infrastructure, Then Use It



An SADT - Level 0 view of domain engineering

Figure C.24-2 An SADT - Level 0 View of Domain Engineering

Arango identifies three distinct models, or forms, of reuse in [ARA88]:

1. *First-order reuse* -- "the desired software system is assembled using preexisting program components. The reusable components become part of the solution, as in the case of reuse of libraries of programs." Arango focuses on first-order reuse in [ARA87].

2. *Second-order reuse* -- "the components reused do not become part of the solution,k but are used to generate the solution. The reuser creates a plan for the construction of a solution out of plan components or by adapting a preexisting plan, as in the case of software implementation using formal transformations."

3. *Third-order reuse* -- 'presumes that meaningful relationships between specifications and the goals of the software construction process can be made explicit and reused. The generation of plans to satisfy appropriate goals has been demonstrated in some specialized program synthesis systems."

Arango warns that the choice between *shallow* or *deep* task models has important methodological and practical consequences. Given their short term goal, they chose the "convenience of exploring heuristic or shallow models, because they can be implemented as deductive systems and mechanized using existing technologies."

Domain as restriction - applying the notion of domain to structuring and restricting the search space of a reuser performing a reuse task. Such an application can make the reuse task tractable and its mechanization efficient.

This is the approach used by Draco, where the domain is **not** the set of transformations, but a restriction on the search-space of Draco which defines the set of transformations that can be applied.

Problem domain - items of real-world information related with respect to some class of significant and solvable problems. While specific problem domain definitions are community-dependent, Arango hypothesizes that there "exists a unified linguistic framework to describe a problem domain which defines a consistent set of perceptions of the community."

Arango notes that the selection of applications as belonging to a particular problem domain "is often based on economic and managerial reasons." He cites both the CAMP effort and the DoD Reusability Guidebook as representative of this approach.

Model of the Domain (MoD) - "a DE-level structure such that:

1. explicitly captures a task-dependent ontology of the problem domain,

2. the competence and performance of the reuser can be assessed on the basis of the structure of the model,

3. improvements in the competence and performance of the reuser can be defined in terms of (DE-level) operations on the structure of the model, and

4. the structure of the model is sufficient to guide the process of abstraction acquisition and provides the means for evolution through formal restructuring."

DE Methodology - a body of principles, methods, representations, and tools which guides and supports (a) the definition, construction, analysis, and controlled evolution of MoDs; and (b) the projection of MoDs onto reuser-dependent encapsulation formalisms.

Note that Arango considers the design of encapsulation mechanisms to be outside the scope of a domain engineering paradigm.

MoD Analysis Method - a collection of analysis and evaluation procedures that are applied to a MoD to identify opportunities for improving the competence and performance of the reuser.

Relative Completeness of a MoD - given a set of problems P and a model R of the reuse task, a MoD is said to be complete relative to P and R if for all problems similar to problems in P, a mechanical reuser driven by R can derive at least one satisfying solution.

Adequate Completeness of a MoD - given a pattern of reuse P (a collection of problems) a MoD is said to be adequately complete with respect to P if the cost $R_f(P)$ over a given period of time is less than the cost of restructuring the MoD.

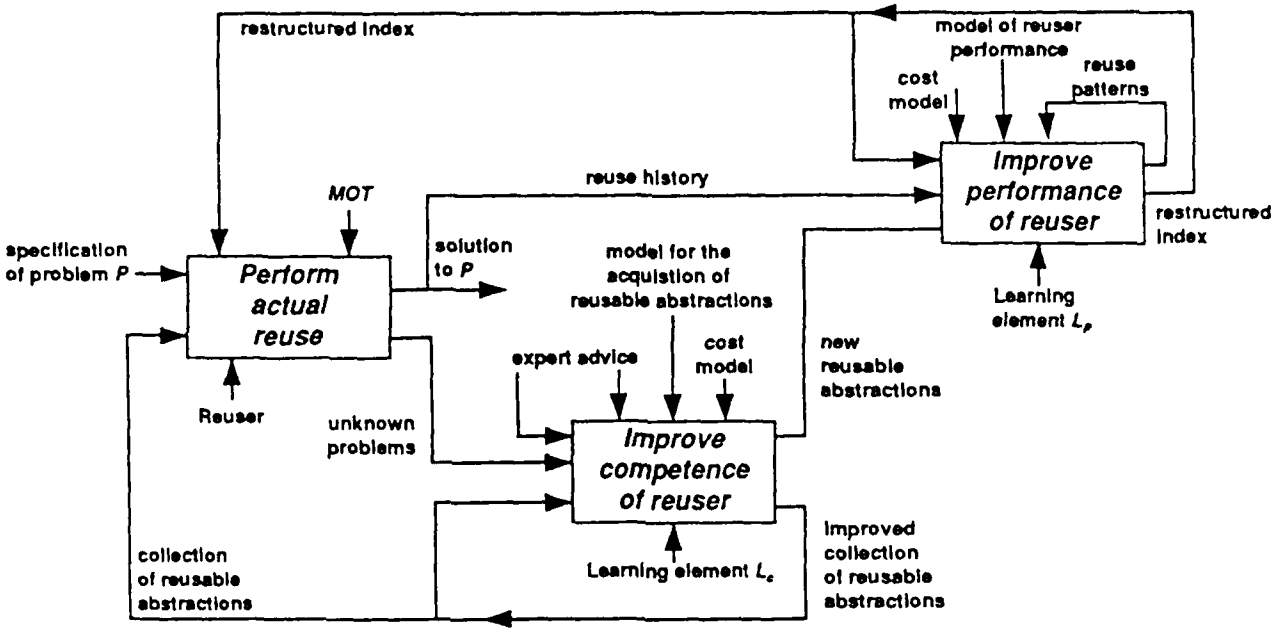
"Evolution is an essential concern in modeling collections of reusable abstractions. A collection of reusable abstractions is bound to evolve for several reasons: natural evolution of the problem domain, our partial understanding of the problem domain, shifting patterns of reuse, changing encapsulation technologies."

Arango proposes a model for *adaptative reuse* as illustrated in Figure C.24-3. He concedes that this model represents "an ideal state of affairs. Continuous analysis and adaptation of a MoD is an expensive process, in particular because it involves the domain engineer and domain experts, the most expensive items in the ... budget."

Arango's team identified the following activities as necessary to bridge the problems-reusable encapsulations gap:

- o Formalization of the reuse paradigm - synthesizing a Model of the Task (MoT)
- o Domain identification - defining the boundaries of a problem domain
- o Abstraction acquisition - including demand-driven knowledge acquisition from experts, as well as model- driven restructuring of the MoD
- o MoD analysis
- o Generation of rationales for reusable abstractions (keeping a history of the genesis of the reusable abstractions)
- o MoD evolution
- o Projection of reusable abstractions onto encapsulation languages

Application of the activities is guided by the DE methodology. Arango briefly presented some requirements for a DE methodology covering general, task-specific and performance considerations. **Reusable abstractions** support the ability to assess the *relevance* of acquired reusable abstractions, and the *relative completeness* of the MoD. Also, they allow for economic



Reuse as an adaptive process

Figure C.24-3 Reuse As An Adaptive Process

projections onto practical encapsulation languages. **Abstraction acquisition** can proceed *incrementally*, and is *model-driven* to avoid the impact of differences among domain engineers. **Analysis methods** allow for a rigorous analysis of the *competence* and *performance* of the reuser. **Evolution methods** can proceed *incrementally*, and can *guarantee improvements* in the competence and performance of the reuser, and preserve the *well-formedness* of the MoD. The DE methodology must be **systematic** such that each method is exhaustive but not redundant. It must a;sps support **documentation** of the MoD's evolution. Methods and representations must be both domain- and encapsulation -independent. These **reuse tasks must be supported:** 1) identification, 2) selection, 3) retrieval, 4) adjustment, and 5) integration of reusable abstractions. **Performance** is considered to provide a *practical* methodology. This may involve approximations or compromises in methods and representations. Lack of competence or performance of the reuser must also be addressed in performing cost assessments and tradeoffs.

Arango cites four parameters as being critical to the success of a domain engineering methodology: the MoT, the MoD, the models of competence and performance of the reuser, and the models of learning adopted. The parameters are interrelated in that the MoT determines the structure of the MoD. The structure of the MoD determines what kinds of analyses of the reuser are possible and what kinds of things can be learned.

The *representation* of information is recognized as an underlying but "fundamental parameter whose choice biases and limits what can be modeled, what can be analyzed, and what can be learned." Finally, the need for mechanization is noted.

Empirical investigations into the following areas were recommended:

1. Determining the performance of DE methodologies

2. Identifying opportunities for mechanizing DE processes and representations

3. Acquiring data to develop cost equations for DE-related decision making

4. Gaining experience in areas like human factors, training, DE project management, and integration of DE with existing practices.

Baxter believes that traditional software engineering has failed to address an important source of information for effective software construction, namely, "knowledge of the problem area". He defines domain analysis to be "processes to capture that knowledge." [BAX88]

"The term 'analysis' is a bit misleading; it implies extracting, from a problem area, the essential concepts. While this is useful for understanding, such concepts need ties to implementation technologies if they are to be used for construction purposes ... Any methodology for capturing problem domain information must also describe how that domain information is used in implementations." The author cites Neighbors "pragmatic approach" (that is, Draco) as an example meeting this constraint. He uses Arango's "term 'domain engineering' to cover the notion of strict domain analysis", separate from the "capture of implementation knowledge."

"One possible methodology for domain engineering used real implementations to guide the acquisition of domain concepts and their implementation ties. A designer uses a transformation system to reverse-engineer an existing application; the domain knowledge is captured in the set of concepts proposed by the designer to be transformed to the final implementation. This approach has been tried with some success on a project to port a moderate-sized LISP program from one machine to another."

"A second insight into domain engineering due to [Neighbor's work] is the need for multiple domains to be used for construction of individual software systems. In some sense, this is simply

knowledge modularization." The author's experience in acquiring such multiple domains was that it was a "hard task; one is not likely to get the investment back on a single application. Thus, the domains have a longer life cycle than the individual systems which use them."

Baxter believes that domains will have more value if alternative configurations of domains can be reassembled depending on the new application class. "This argues for domains having semantics independent of the application, and certainly not defined in an operational (as in object oriented approaches) or transformational (as in Neighbors Draco system) sense." He argues "that each domain have semantics independent of other domains, and independent of the actual implementation." Baxter proposes the use of algebraic representations "along the line of Goguen" as the appropriate domain representation technique. He also notes that support for the configuration process requires that "relationships between domains be established quickly and easily."

The author finds it convenient to store domain implementation knowledge along with the domain semantic knowledge in order that the domain name can serve as an index to both sets of information. He emphasizes, though, that "the semantic knowledge is the fundamental, and the implementation knowledge is secondary." Useful implementation knowledge includes domain transformation concepts, and performance or control knowledge.

Baxter notes several "difficulties with dividing problem area knowledge into individual domains." That is, the process of defining the domain boundaries. "Currently, we know very little about how to classify concepts according to domains, about extending domains to cover unclassifiable concepts, and about when to create a new domain." He feels that an algebraic approach provides clues for how to combine "fragments" of independent domains into meaningful "compositions".

"Clearly, domain 'analysis' is in its infancy, even as an art."

Domain analysis is a key aspect of the Draco method and toolset. Draco is based on the concept of performing a domain analysis to determine a common set of objects and operations performed on them for a particular domain (the domain model). A custom domain language is then defined which deals with those objects and operations identified in the domain analysis. Once the language is defined, Draco transforms the specification into an executable program. [NEI81] [NEI83]

Draco hypothesizes two new user roles in the development of software: the domain analyst and the domain designer. The domain analyst examines the requirements of the set of similar systems in a given problem area. The domain designer specifies identified by the domain analyst. These implementations are specified in terms of objects and operations found in other domains already known to Draco. "Traditional" systems analysts and designers interact with Draco in building systems from the existing domain information. The use of the Draco tool also requires the assistance of a Draco systems specialist who defines the low-level transformations to program code. [NEI83] Figure C.24-4 illustrates the organizational context for Draco's use.

Neighbors defines the following approach to creating a software system with Draco [NEI81] (Figures C.24-5 to C.24-11 provide the author's SADT diagrams illustrating the activities and data involved):

o Determine the domains of interest to the organization

o Analyze the domain

- Correlate all available information

- Determine if the information is sufficient. If so,

o Construct the domain

- Create a parser describing the syntax and internal form of the domain

- Create a "pretty-printer" describing the external form of the domain as shown to users

- Create a library of transformations representing the rules of exchange between objects and operations in the domain in terms of language translations

- Create a component library specifying the semantics of the domain. There is one software component for each object and operation in the domain, each having several possible refinements

- Define domain-specific procedures for optimizing Draco transformations as required

o Test the domain. If successful,

o Add the domain to the library of domains known to Draco

o Construct a software system

- Analyze the requirements of the system under consideration

- Analyze the past performance of the Draco domain

- Decide whether to construct a custom system or a Draco system (or both)

- Build the system as appropriate. Using Draco,

+ Choose a domain

+ Choose an instance

+ Choose a locale

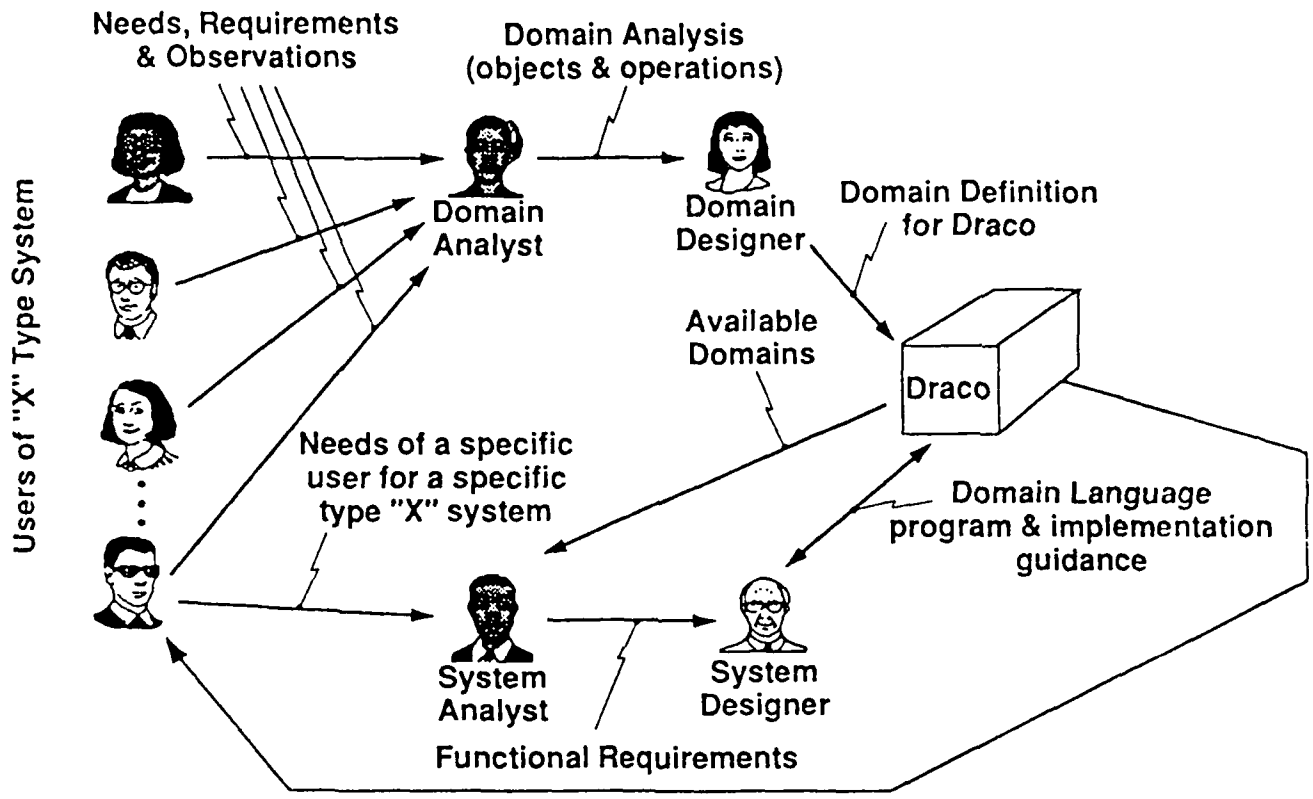
+ Transform the domain

+ Refine the domain

o Establish new Draco domains or revise existing domains based on the experience

Draco applies the domain concept throughout the software generation process as an aid in reducing complexity. Starting with the problem domain, the system designer deals with various modeling domains during the refinement process until the final target domain is reached. "The concept of domain here is very useful in supplying a psychological set to the system's designer (i.e., the designer must only consider and think about the objects and operations of one domain at a time)." [NEI83]

"Only about 10 or 12 fully usable Draco domains have been built and each has reinforced the idea that domain analysis and design is **very hard**." [NEI83] Neighbors emphasizes the need for an *expert* to perform the domain analysis, and the need for well-documented work to be able to extract an appropriate set of objects and operations. He noted that developing domain syntaxes remains a creative effort, and the resulting domain specifications can be very lengthy.



Organizational Context of Draco

Figure C.24-4 Organizational Context of Draco

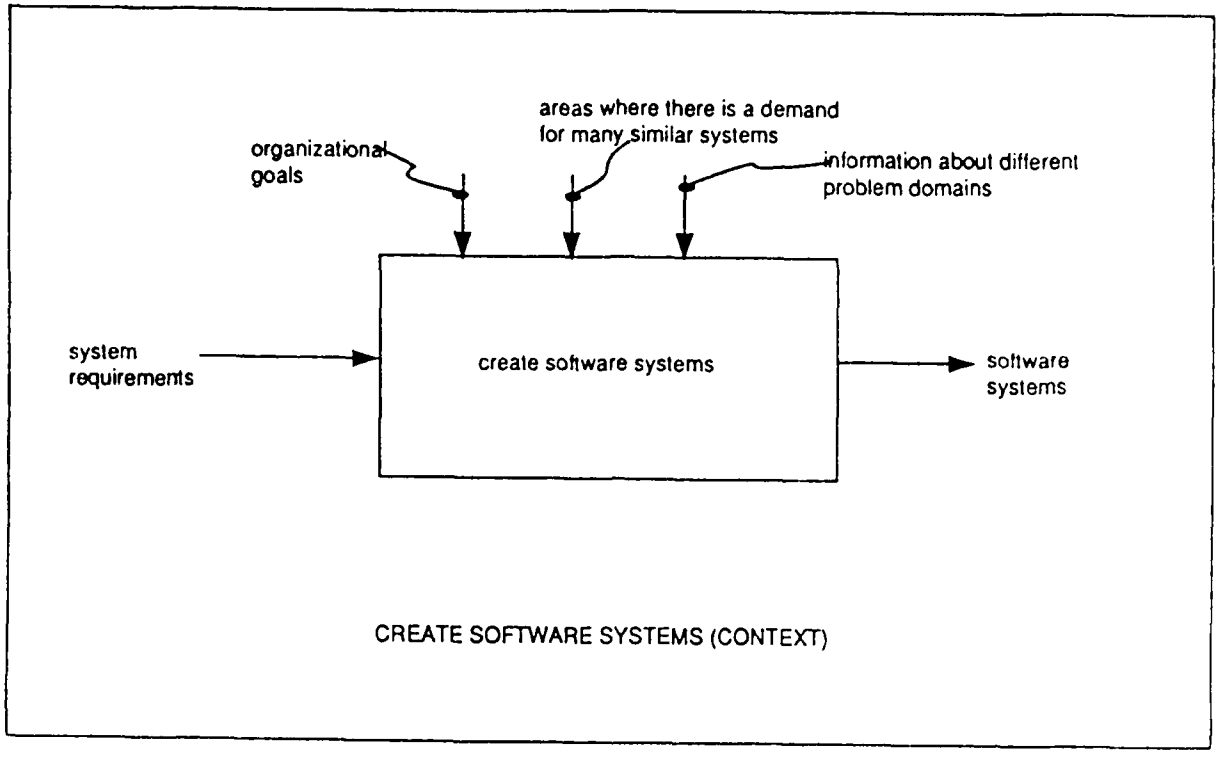


Figure C.24-5 Create Software Systems (Context)

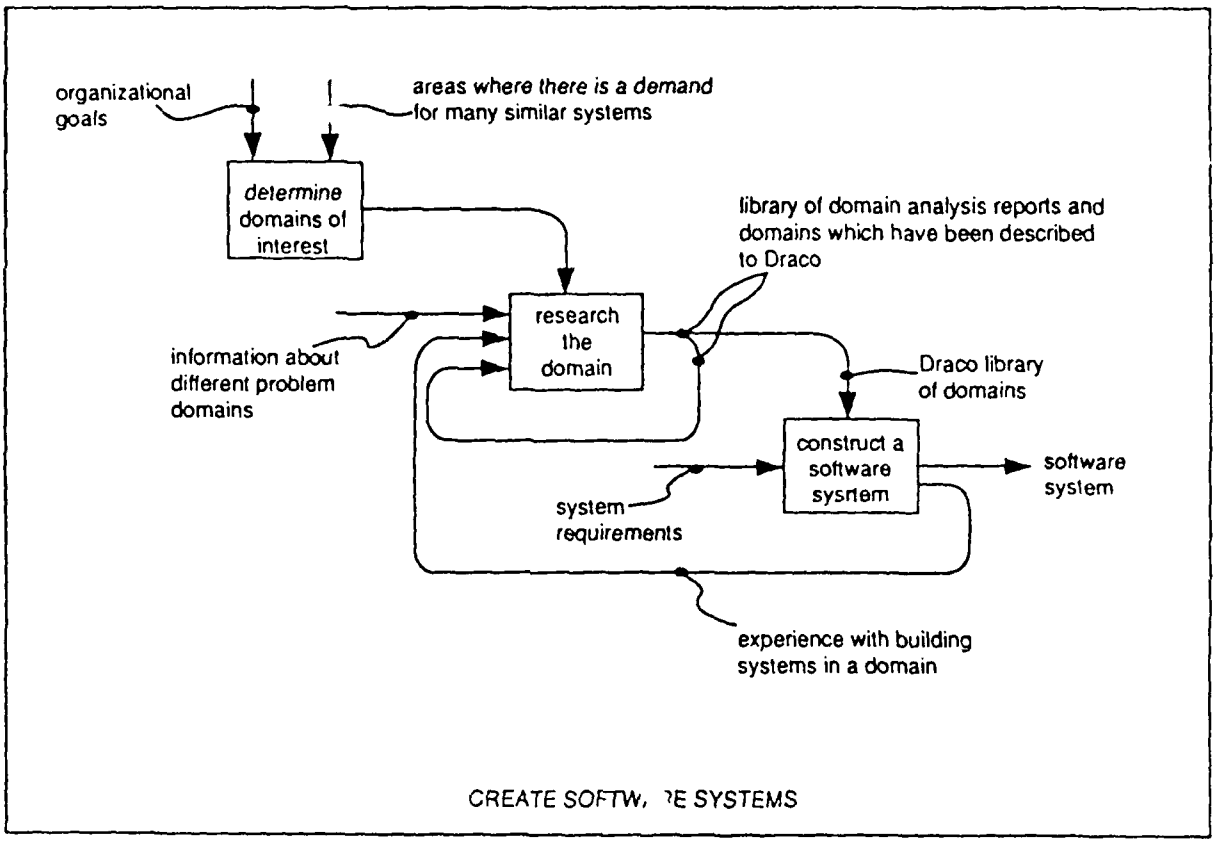


Figure C.24-6 Create Software Systems

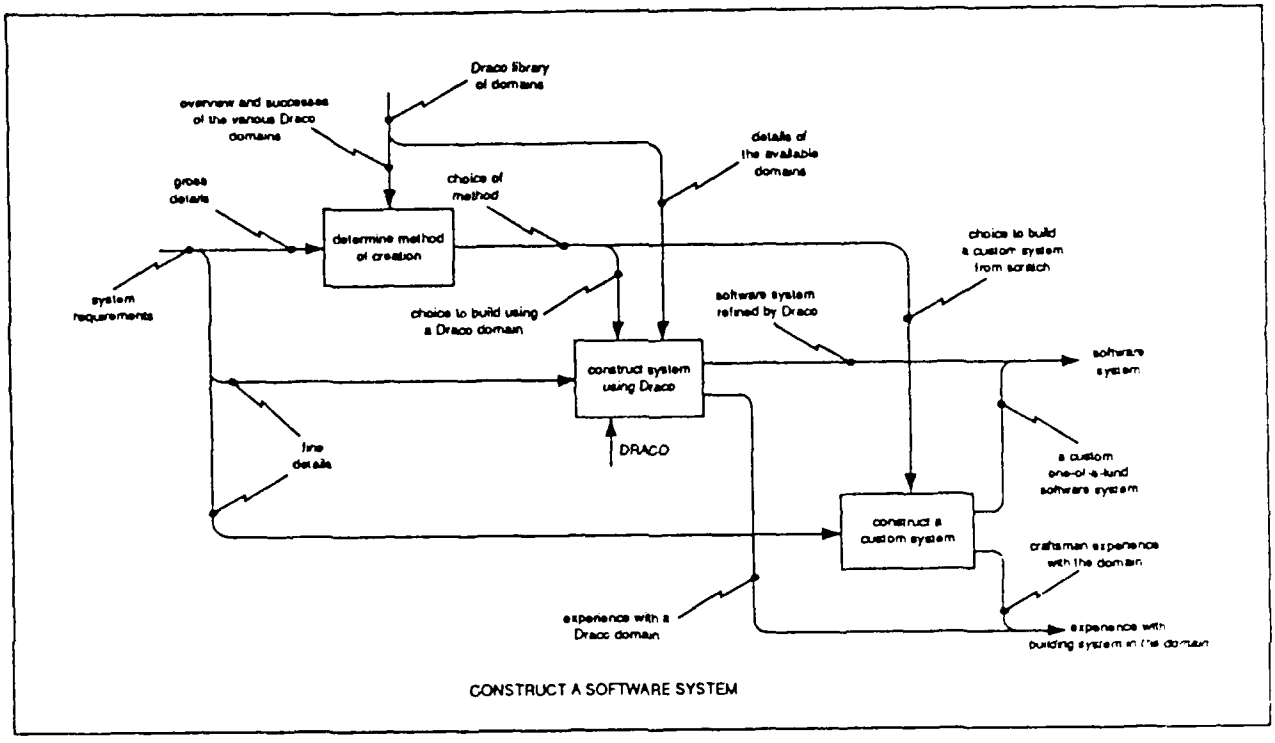


Figure C.24-7 Construct a Software System

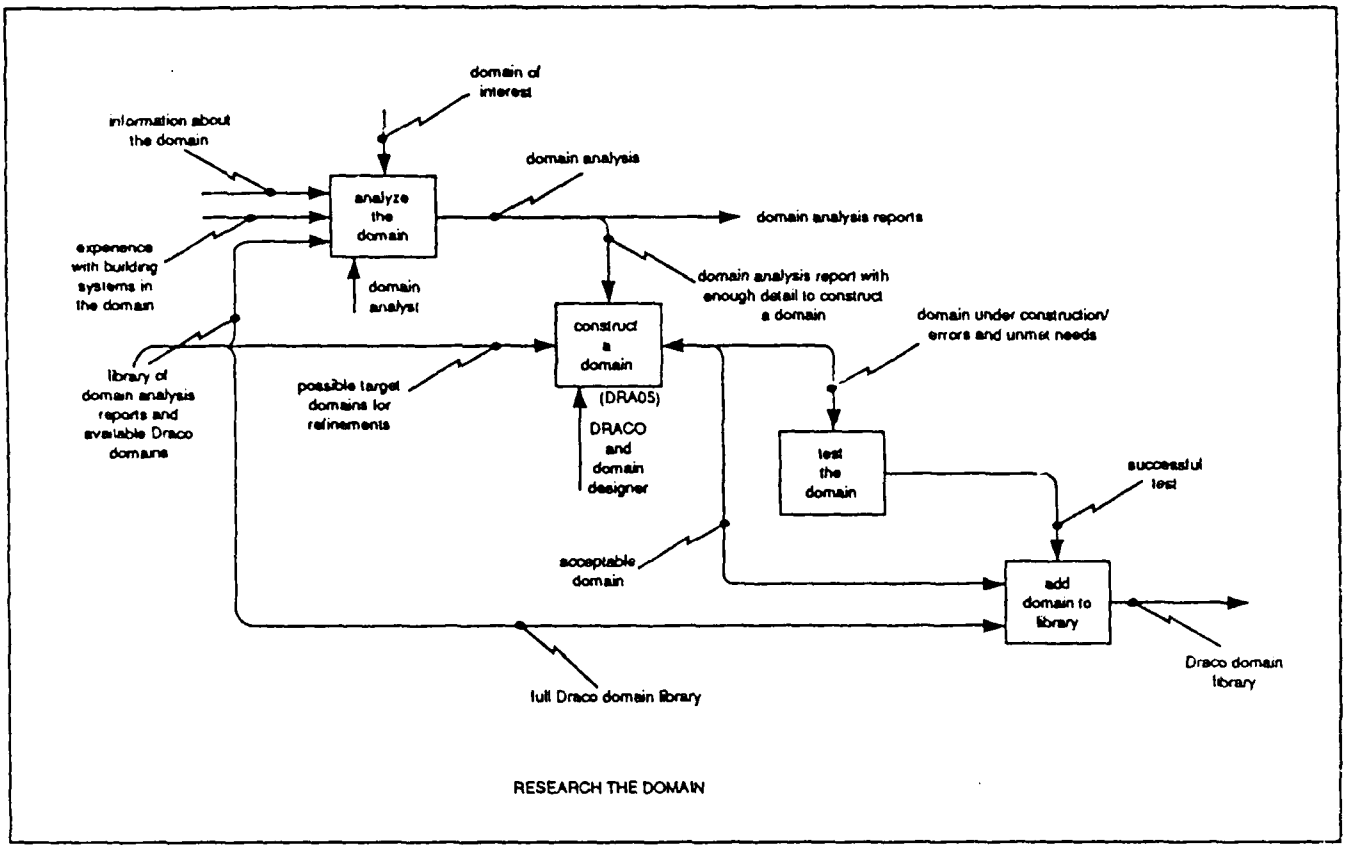


Figure C.24-8 Research the Domain

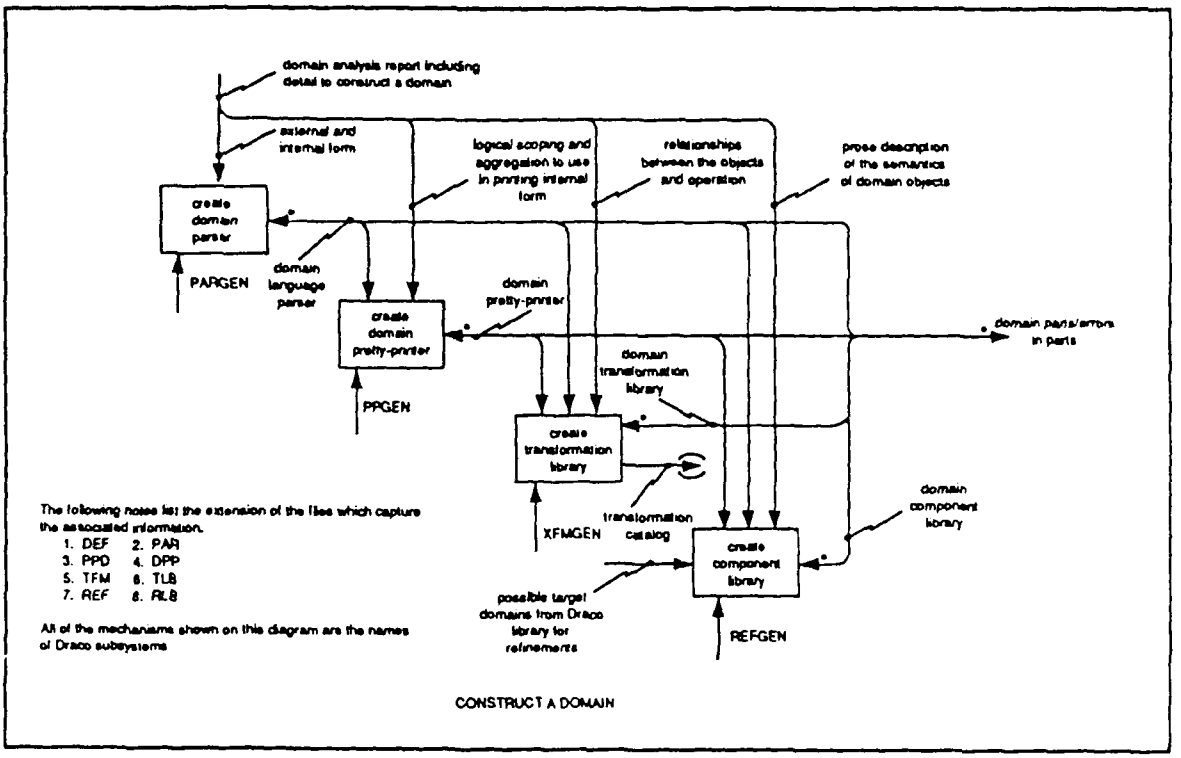


Figure C.24-9 Construct a Domain

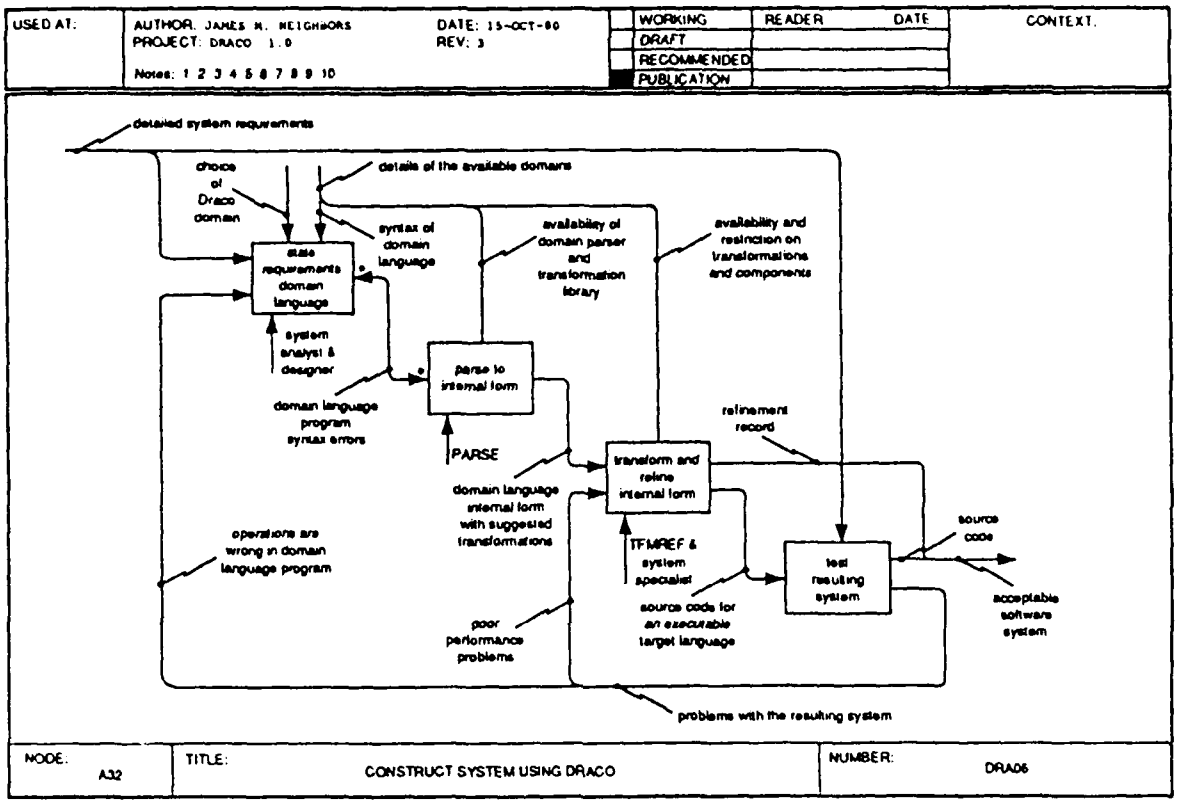


Figure C.24-10 Construct System Using Draco

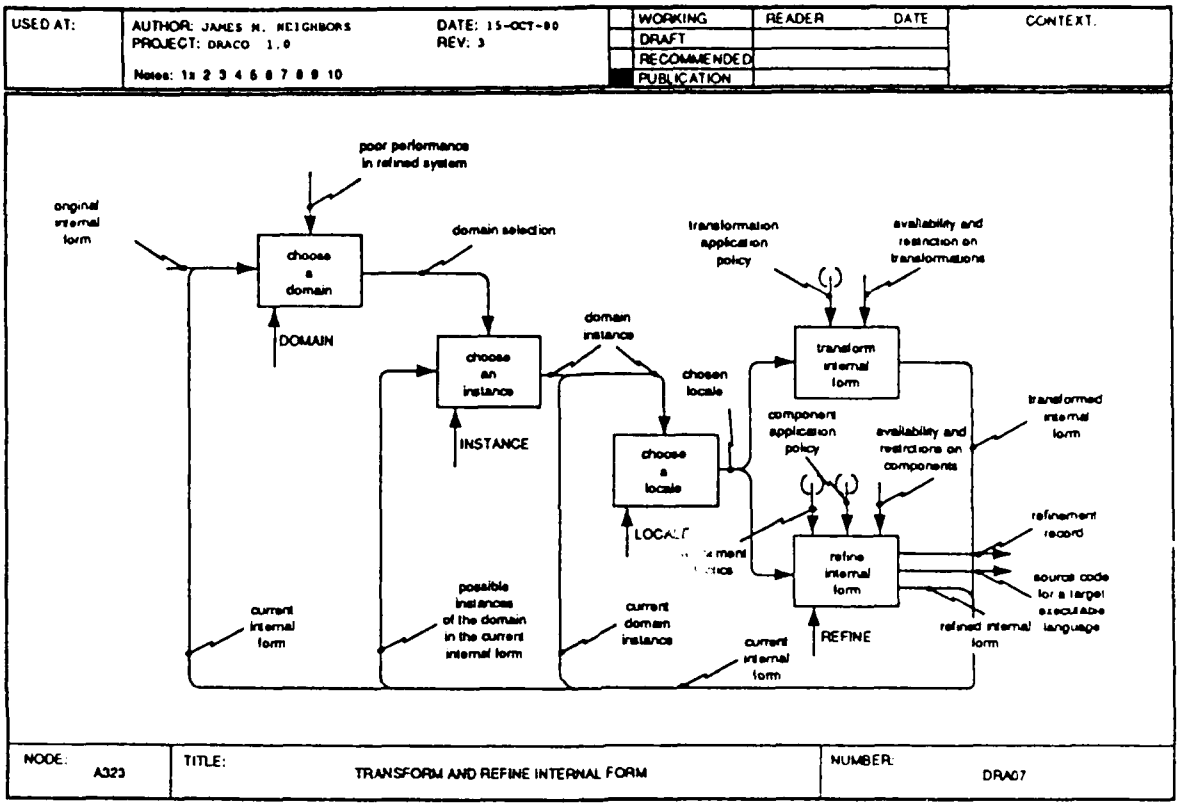


Figure C.24-11 Transform and Refine Internal Form

Freeman [FRE87] analyzed how the mechanism of Draco contributed to software reusability, and concluded:

"One of the goals of reusability is to make it possible to reuse the understanding of a problem or problem domain developed during the analysis portion of development. The languages that can be created using Draco permit the capture and reuse of information about an entire problem domain. Specifically, the languages capture the domain analyst's understanding of which operators and objects are appropriate for describing operations in a particular domain. Later, when a system analyst picks up that domain language to use it, he or she is then reusing the analysis of the domain originally developed by the domain analyst."

C.25 University of Oregon

In [ROB88], Robinson addresses the use of domain models representing domain goals and their relationships. Robinson recognizes that such a model is not sufficient nor adequate representation of domain information. However, he focuses on the importance of at least this piece of the total information for three purposes:

"(1) justification of the inclusion/exclusion of specification components

(2) compromises of conflicting specification components

(3) 'novel' resolutions of conflicting components."

Robinson proposes a methodology, called Parallel Elaboration of Specifications (PES), which supports *multi-perspective design*. He feels that the ability of PES to support an analysis of consistency of a specification "across development lines", but not requiring such consistency, is one major advantage of PES over other methodologies.

"Multi-perspective design calls for the development of whole specifications from the desires of a subset of system agents", that is, from the perspective of various users, called "selfish views." The specifications are expressed as domain goals and relationships between those goals. Goals are roughly akin to functional and performance requirements.

As one might expect, multiple selfish view specifications are likely to be conflicting, and several techniques for resolving conflicts are explored by the paper. Robinson "suspect[s] that such compromises are commonly done prior to specification, but go unrecorded. By allowing for (in fact pushing for) explicit negotiation of domain goals, we make explicit the intents of the specifier and link those intents to the specification. Hence, we prefer 'selfish views' over functional decomposition due to its predilection to force negotiation of issues."

The resolution of view conflicts via goal integration operators can support "reuse via integration of two (or more) nearly 'correct' systems, i.e., specifications which nearly satisfy the requirements."

The first activity which is performed in PES is the definition of a "base domain model" whose goal attributes are all null. "Refinements of the base model (perspectives) set goal attributes based on the context of the application." From the perspectives, actual specifications are then derived (the goals are operationalized in the specification language). Links between the goals and specifications are maintained. "These links will be used ... to trace specification level conflicts to goal level conflicts during the process of integration." Integration is a three-step process as follows:

- **Correspondence Identification** - identify two specification entities as [non]equivalent; this task must be performed by the user
- **Conflict Detection** - identify which equivalent entities are conflicting; some of this task can be automated; the conflicts are grouped together into abstract classes and then matched against possible resolutions to the conflicts (Note: a common type of 'conflict' to find is simply that two entities are at different levels of abstraction)
- **Conflict Resolution** - remove differences in conflicting specifications using compromise or goal substitution.